

O'REILLY®

C# 12

Справочник

Полное описание языка



Джозеф Албахари

C# 12

Справочник

Полное описание языка

C# 12 IN A NUTSHELL

THE DEFINITIVE REFERENCE

Joseph Albahari

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

C# 12

Справочник

Полное описание языка

Джозеф Албахари



Москва · Санкт-Петербург
2024

ББК 32.973.26-018.2.75

A45

УДК 004.432

ООО “Диалектика”

Перевод с английского и редакция Ю.Н. Артеменко

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:
info.dialektika@gmail.com, <http://www.dialektika.com>

Албахари, Джозеф.

A45 С# 12. Справочник. Полное описание языка. : Пер. с англ. — СПб. : ООО “Диалектика”, 2024. — 1104 с. : ил. — Парал. тит. англ.

ISBN 978-5-907705-47-0 (рус.)

ББК 32.973.26-018.2.75

Все права защищены.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly Media, Inc.

Copyright © 2024 Joseph Albahari.

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Authorized translation from the English language edition of the *C# 12 in a Nutshell: The Definitive Reference* (ISBN 978-1-098-14744-0), published by O'Reilly Media, Inc.

No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Джозеф Албахари

С# 12. Справочник. Полное описание языка

Выпускающий редактор А.С. Тополенко

Ответственный редактор М.С. Репин

Главный дизайнер А.Г. Корнейчик

Корректор Н.С. Войтенко

Рецензент Н.К. Репина

Редактор М.В. Дубовцева

Подписано в печать 14.08.2024. Формат 70×100/16

Усл. печ. л. 89,01. Уч.-изд. л. 63,2

Доп. тираж 400 экз. Заказ № 4873.

Отпечатано в АО “Первая Образцовая типография”

Филиал “Чеховский Печатный Двор”

142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(495)107-02-68

ООО “Диалектика”, 195027, г. Санкт-Петербург, ул. Магнитогорская, д. 30, литер А, пом. 828А, п/я 116

ISBN 978-5-907705-47-0 (рус.)

© ООО “Диалектика”, 2024,
перевод, оформление, макетирование

© 2024 Joseph Albahari

ISBN 978-1-098-14744-0 (англ.)

Оглавление

Предисловие	29
Глава 1. Введение в C# и .NET	35
Глава 2. Основы языка C#	67
Глава 3. Создание типов в языке C#	139
Глава 4. Дополнительные средства языка C#	211
Глава 5. Обзор .NET	317
Глава 6. Основы .NET	331
Глава 7. Коллекции	407
Глава 8. Запросы LINQ	463
Глава 9. Операции LINQ	517
Глава 10. LINQ to XML	569
Глава 11. Другие технологии XML и JSON	601
Глава 12. Освобождение и сборка мусора	631
Глава 13. Диагностика	659
Глава 14. Параллелизм и асинхронность	683
Глава 15. Потоки данных и ввод-вывод	749
Глава 16. Взаимодействие с сетью	797
Глава 17. Сборки	823
Глава 18. Рефлексия и метаданные	865
Глава 19. Динамическое программирование	921
Глава 20. Криптография	935
Глава 21. Расширенная многопоточность	949
Глава 22. Параллельное программирование	993
Глава 23. <code>Span<T></code> и <code>Memory<T></code>	1039
Глава 24. Способность к взаимодействию	1051
Глава 25. Регулярные выражения	1077
Предметный указатель	1098

Содержание

Об авторе	28
Предисловие	29
Предполагаемая читательская аудитория	29
Как организована эта книга	30
Что требуется для работы с этой книгой	30
Соглашения, используемые в этой книге	31
Использование примеров кода	32
Ждем ваших отзывов!	32
Благодарности	33
Глава 1. Введение в C# и .NET	35
Объектная ориентация	35
Безопасность в отношении типов	36
Управление памятью	37
Поддержка платформ	37
Общязыковые исполняющие среды, библиотеки базовых классов и исполняющие среды	38
Общязыковая исполняющая среда	38
Библиотека базовых классов	39
Исполняющие среды	39
Нишевые исполняющие среды	42
Краткая история языка C#	43
Нововведения версии C# 12	43
Нововведения версии C# 11	45
Нововведения версии C# 10	47
Нововведения версии C# 9.0	50
Нововведения версии C# 8.0	53
Нововведения версий C# 7.x	57
Нововведения версии C# 6.0	61
Нововведения версии C# 5.0	63
Нововведения версии C# 4.0	63
Нововведения версии C# 3.0	64
Нововведения версии C# 2.0	65
Глава 2. Основы языка C#	67
Первая программа на C#	67
Компиляция	68
Синтаксис	70
Идентификаторы и ключевые слова	70
Литералы, знаки пунктуации и операции	71
Комментарии	72
Основы типов	72
Примеры предопределенных типов	72

Специальные типы	73
Типы и преобразования	78
Типы значений и ссылочные типы	78
Классификация предопределенных типов	82
Числовые типы	83
Числовые литералы	84
Числовые преобразования	85
Арифметические операции	86
Операции инкремента и декремента	86
Специальные операции с целочисленными типами	87
8- и 16-битные целочисленные типы	89
Специальные значения float и double	89
Выбор между double и decimal	90
Ошибки округления вещественных чисел	91
Булевский тип и операции	91
Булевые преобразования	91
Операции сравнения и проверки равенства	92
Условные операции	92
Строки и символы	93
Символьные преобразования	94
Строчный тип	94
Строки UTF-8	97
Массивы	98
Стандартная инициализация элементов	99
Индексы и диапазоны	99
Многомерные массивы	100
Упрощенные выражения инициализации массивов	102
Проверка границ	103
Переменные и параметры	103
Стек и куча	103
Определенное присваивание	104
Стандартные значения	105
Параметры	106
Локальные ссылочные переменные	111
Возвращаемые ссылочные значения	112
Объявление неявно типизированных локальных переменных с помощью var	113
Выражения new целевого типа	114
Выражения и операции	114
Первичные выражения	115
Пустые выражения	115
Выражения присваивания	115
Приоритеты и ассоциативность операций	115
Таблица операций	116
Операции для работы со значениями null	116
Операция объединения с null	120

Операция присваивания с объединением с null	120
null-условная операция	120
Операторы	121
Операторы объявления	122
Операторы выражений	122
Операторы выбора	123
Операторы итераций	128
Операторы перехода	130
Смешанные операторы	131
Пространства имен	132
Пространства имен с областью видимости на уровне файлов	133
Директива using	133
Директива global using	133
Директива using static	134
Правила внутри пространства имен	134
Назначение псевдонимов типам и пространствам имен	136
Дополнительные возможности пространств имен	137
Глава 3. Создание типов в языке C#	139
Классы	139
Методы	142
Конструкторы экземпляров	144
Деконструкторы	146
Инициализаторы объектов	147
Ссылка this	149
Свойства	149
Индексаторы	154
Основные конструкторы (C# 12)	155
Статические конструкторы	159
Статические классы	160
Финализаторы	160
Частичные типы и методы	161
Операция nameof	162
Наследование	163
Полиморфизм	164
Приведение и ссылочные преобразования	164
Виртуальные функции-члены	167
Абстрактные классы и абстрактные члены	169
Скрытие унаследованных членов	169
Запечатывание функций и классов	170
Ключевое слово base	170
Конструкторы и наследование	171
Перегрузка и распознавание	174
Тип object	174
Упаковка и распаковка	175

Статическая проверка типов и проверка типов во время выполнения	176
Метод <code>GetType</code> и операция <code>typeof</code>	177
Метод <code>ToString</code>	177
Список членов <code>object</code>	178
Структуры	178
Семантика конструирования структур	179
Ссылачные структуры	180
Модификаторы доступа	181
Примеры	182
Дружественные сборки	183
Установление верхнего предела доступности	183
Ограничения, накладываемые на модификаторы доступа	183
Интерфейсы	184
Расширение интерфейса	185
Явная реализация членов интерфейса	185
Реализация виртуальных членов интерфейса	186
Повторная реализация члена интерфейса в подклассе	187
Интерфейсы и упаковка	188
Стандартные члены интерфейса	188
Статические члены интерфейса	189
Перечисления	191
Преобразования перечислений	192
Перечисления флагов	192
Операции над перечислениями	193
Проблемы безопасности типов	193
Вложенные типы	195
Обобщения	196
Обобщенные типы	196
Для чего предназначены обобщения	197
Обобщенные методы	198
Объявление параметров типа	199
Операция <code>typeof</code> и несвязанные обобщенные типы	200
Стандартное значение для параметра обобщенного типа	200
Ограничения обобщений	201
Создание подклассов для обобщенных типов	203
Самоссылающиеся объявления обобщений	203
Статические данные	204
Параметры типа и преобразования	204
Ковариантность	205
Контравариантность	208
Сравнение обобщений C# и шаблонов C++	209
Глава 4. Дополнительные средства языка C#	211
Делегаты	211
Написание подключаемых методов с помощью делегатов	212

Целевые методы экземпляра и статические целевые методы	213
Групповые делегаты	213
Обобщенные типы делегатов	215
Делегаты Func и Action	215
Сравнение делегатов и интерфейсов	216
Совместимость делегатов	217
События	219
Стандартный шаблон событий	221
Средства доступа к событию	224
Модификаторы событий	226
Лямбда-выражения	226
Явное указание типов параметров и возвращаемого типа лямбда-выражения	227
Стандартные параметры лямбда-выражений (C# 12)	228
Захватывание внешних переменных	228
Сравнение лямбда-выражений и локальных методов	231
Анонимные методы	232
Операторы try и исключения	233
Конструкция catch	235
Блок finally	236
Генерация исключений	238
Основные свойства класса System.Exception	239
Общие типы исключений	240
Шаблон методов TryXXX	241
Альтернативы исключениям	241
Перечисление и итераторы	242
Перечисление	242
Инициализаторы и выражения коллекций	243
Итераторы	244
Семантика итератора	245
Компоновка последовательностей	246
Типы значений, допускающие null	247
Структура Nullable<T>	248
Подъем операций	249
Тип bool? и операции & и	251
Типы значений, допускающие null, и операции для работы с null	251
Сценарии использования типов значений, допускающих null	252
Альтернативы типам значений, допускающим null	252
Ссылочные типы, допускающие null	253
null-терпимая операция	254
Разъединение контекстов с заметками и с предупреждениями	255
Трактовка предупреждений о допустимости значения null как ошибок	256
Расширяющие методы	256
Цепочки расширяющих методов	257
Неоднозначность и распознавание	257
Анонимные типы	259

Кортежи	260
Именование элементов кортежа	261
Назначение псевдонимов кортежам (C# 12)	263
Метод <code>ValueTuple.Create</code>	263
Деконструирование кортежей	264
Сравнение эквивалентности	264
Классы <code>System.Tuple</code>	265
Записи	265
Подоплека	265
Определение записи	266
Неразрушающее изменение	270
Проверка достоверности свойств	271
Вычисляемые поля и ленивая оценка	272
Основные конструкторы	274
Записи и сравнение эквивалентности	276
Шаблоны	277
Шаблон константы	278
Реляционные шаблоны	278
Комбинаторы шаблонов	279
Шаблон <code>var</code>	279
Шаблоны кортежей и позиционные шаблоны	279
Шаблоны свойств	280
Шаблоны списков	282
Атрибуты	282
Классы атрибутов	283
Именованные и позиционные параметры атрибутов	283
Применение атрибутов к сборкам и поддерживающим полям	284
Применение атрибутов к лямбда-выражениям	284
Указание нескольких атрибутов	285
Атрибуты информации о вызывающем компоненте	285
Атрибут <code>CallerArgumentExpression</code>	287
Динамическое связывание	287
Сравнение статического и динамического связывания	288
Специальное связывание	289
Языковое связывание	290
Исключение <code>RuntimeBinderException</code>	290
Представление типа <code>dynamic</code> во время выполнения	291
Динамические преобразования	292
Сравнение <code>var</code> и <code>dynamic</code>	292
Динамические выражения	292
Динамические вызовы без динамических получателей	293
Статические типы в динамических выражениях	294
Невызываемые функции	294
Перегрузка операций	296
Функции операций	296

Перегрузка операций эквивалентности и сравнения	298
Специальные неявные и явные преобразования	298
Перегрузка операций <code>true</code> и <code>false</code>	299
Статический полиморфизм	300
Полиморфные операции	301
Обобщенная математика	302
Небезопасный код и указатели	303
Основы указателей	303
Небезопасный код	303
Оператор <code>fixed</code>	304
Операция указателя на член	304
Ключевое слово <code>stackalloc</code>	305
Буферы фиксированных размеров	305
<code>void*</code>	306
Целочисленные типы с собственным размером	306
Указатели на функции	308
Атрибут <code>SkipLocalsInit</code>	309
Директивы препроцессора	310
Условные атрибуты	311
Директива <code>#pragma warning</code>	312
XML-документация	312
Стандартные XML-дескрипторы документации	313
Дескрипторы, определяемые пользователем	315
Перекрестные ссылки на типы или члены	315
Глава 5. Обзор .NET	317
Целевые платформы и TFM	319
.NET Standard	319
.NET Standard 2.0	320
Другие стандарты .NET Standard	320
Совместимость .NET Framework и .NET 8	320
Ссылочные сборки	321
Версии исполняющих сред и языка C#	321
Среда CLR и библиотека BCL	322
Системные типы	322
Обработка текста	322
Коллекции	322
Запросы	323
XML и JSON	323
Диагностика	323
Параллелизм и асинхронность	324
Потоки данных и ввод-вывод	324
Работа с сетями	324
Сборки, рефлексия и атрибуты	324
Динамическое программирование	325
Криптография	325

Расширенная многопоточность	325
Параллельное программирование	325
Span<T> и Memory<T>	325
Возможность взаимодействия с собственным кодом и COM	326
Регулярные выражения	326
Сериализация	326
Компилятор Roslyn	326
Прикладные слои	326
ASP.NET Core	327
Windows Desktop	328
MAUI	330
Глава 6. Основы .NET	331
Обработка строк и текста	331
Тип char	331
Тип string	333
Сравнение строк	337
Класс StringBuilder	340
Кодировка текста и Unicode	341
Дата и время	344
Структура TimeSpan	345
Структуры DateTime и DateTimeOffset	346
Структуры DateOnly и TimeOnly	352
Даты и часовые пояса	352
Структура DateTime и часовые пояса	353
Структура DateTimeOffset и часовые пояса	353
Класс TimeZoneInfo	354
Летнее время и структура DateTime	357
Форматирование и разбор	358
Методы ToString и Parse	358
Поставщики форматов	359
Стандартные форматные строки и флаги разбора	364
Форматные строки для чисел	364
Перечисление NumberStyles	367
Форматные строки для даты/времени	368
Перечисление DateTimeStyles	370
Форматные строки для перечислений	371
Другие механизмы преобразования	371
Класс Convert	372
Класс XmlConvert	373
Преобразователи типов	374
Класс BitConverter	375
Глобализация	375
Контрольный перечень глобализации	376
Тестирование	376

Работа с числами	377
Преобразования	377
Класс Math	377
Структура BigInteger	378
Структура Half	379
Структура Complex	380
Класс Random	380
Класс BitOperations	382
Перечисления	382
Преобразования для перечислений	382
Перечисление значений перечисления	385
Как работают перечисления	385
Структура Guid	386
Сравнение эквивалентности	386
Эквивалентность значений и ссылочная эквивалентность	387
Стандартные протоколы эквивалентности	388
Эквивалентность и специальные типы	392
Сравнение порядка	398
Интерфейсы IComparable	398
Операции > и <	399
Реализация интерфейсов IComparable	400
Служебные классы	401
Класс Console	401
Класс Environment	402
Класс Process	402
Класс ApplicationContext	405
Глава 7. Коллекции	407
Перечисление	408
Интерфейсы IEnumerable и IEnumerator	408
Интерфейсы IEnumerable<T> и IEnumerator<T>	409
Реализация интерфейсов перечисления	412
Интерфейсы ICollection и IList	415
Интерфейсы ICollection<T> и ICollection	416
Интерфейсы IList<T> и IList	417
Интерфейсы IReadOnlyCollection<T> и IReadOnlyList<T>	418
Класс Array	419
Конструирование и индексация	422
Перечисление	423
Длина и ранг	424
Поиск	424
Сортировка	425
Обращение порядка следования элементов	427
Копирование	427
Преобразование и изменение размера	427

Списки, очереди, стеки и наборы	428
Классы <code>List<T></code> и <code>ArrayList</code>	428
Класс <code>LinkedList<T></code>	431
Классы <code>Queue<T></code> и <code>Queue</code>	432
Классы <code>Stack<T></code> и <code>Stack</code>	433
Класс <code>BitArray</code>	434
Классы <code>HashSet<T></code> и <code>SortedSet<T></code>	435
Словари	436
Интерфейс <code>IDictionary< TKey, TValue ></code>	437
Интерфейс <code>IDictionary</code>	439
Классы <code>Dictionary< TKey, TValue ></code> и <code>Hashtable</code>	439
Класс <code>OrderedDictionary</code>	441
Классы <code>ListDictionary</code> и <code>HybridDictionary</code>	442
Отсортированные словари	442
Настраиваемые коллекции и посредники	444
Классы <code>Collection<T></code> и <code>CollectionBase</code>	444
Классы <code>KeyedCollection< TKey, TItem ></code> и <code>DictionaryBase</code>	446
Класс <code>ReadOnlyCollection<T></code>	449
Неизменяемые коллекции	450
Создание неизменяемых коллекций	451
Манипулирование неизменяемыми коллекциями	451
Построители	452
Неизменяемые коллекции и производительность	452
Замороженные коллекции	453
Подключение протоколов эквивалентности и порядка	454
Интерфейсы <code>IEqualityComparer</code> и <code>EqualityComparer</code>	455
Интерфейс <code>IComparer</code> и класс <code>Comparer</code>	457
Класс <code>StringComparer</code>	459
Интерфейсы <code>IStructuralEquatable</code> и <code>IComparable</code>	460
Глава 8. Запросы LINQ	463
Начало работы	463
Текущий синтаксис	466
Выстраивание в цепочки операций запросов	466
Составление лямбда-выражений	468
Естественный порядок	471
Другие операции	471
Выражения запросов	472
Переменные диапазона	474
Сравнение синтаксиса запросов и синтаксиса SQL	475
Сравнение синтаксиса запросов и текущего синтаксиса	475
Запросы со смешанным синтаксисом	476
Отложенное выполнение	476
Повторное вычисление	477
Захваченные переменные	478
Как работает отложенное выполнение	479

Построение цепочки декораторов	480
Каким образом выполняются запросы	481
Подзапросы	482
Подзапросы и отложенное выполнение	485
Стратегии композиции	486
Постепенное построение запросов	486
Ключевое слово <code>into</code>	487
Упаковка запросов	488
Стратегии проецирования	490
Инициализаторы объектов	490
Анонимные типы	490
Ключевое слово <code>let</code>	491
Интерпретируемые запросы	492
Каким образом работают интерпретируемые запросы	494
Комбинирование интерпретируемых и локальных запросов	496
Метод <code>AsEnumerable</code>	497
Инфраструктура EF Core	499
Сущностные классы EF Core	499
Объект <code>DbContext</code>	499
Отслеживание объектов	504
Отслеживание изменений	505
Навигационные свойства	506
Отложенное выполнение	509
Построение выражений запросов	511
Сравнение делегатов и деревьев выражений	511
Деревья выражений	513
Глава 9. Операции LINQ	517
Обзор	518
Последовательность → последовательность	519
Последовательность → элемент или значение	520
Ничего → последовательность	521
Выполнение фильтрации	521
<code>Where</code>	522
<code>Take</code> , <code>TakeLast</code> , <code>Skip</code> , <code>SkipLast</code>	524
<code>TakeWhile</code> и <code>SkipWhile</code>	525
<code>Distinct</code> и <code>DistinctBy</code>	525
Выполнение проецирования	526
<code>Select</code>	526
<code>SelectMany</code>	531
Выполнение соединения	538
<code>Join</code> и <code>GroupJoin</code>	539
Операция <code>Zip</code>	547
Упорядочение	547
<code>OrderBy</code> , <code>OrderByDescending</code> , <code>ThenBy</code> , <code>ThenByDescending</code>	547

Группирование	550
GroupBy	550
Chunk	553
Операции над множествами	554
Concat, Union, UnionBy	554
Intersect, IntersectBy, Except, ExceptBy	555
Методы преобразования	555
OfType и Cast	556
ToArray, ToList, ToDictionary, ToHashSet, ToLookup	558
AsEnumerable и AsQueryable	558
Операции над элементами	559
First, Last, Single	559
ElementAt	560
MinBy и MaxBy	560
DefaultIfEmpty	561
Методы агрегирования	561
Count и LongCount	561
Min и Max	562
Sum и Average	562
Aggregate	563
Квантификаторы	566
Contains и Any	566
All и SequenceEqual	567
Методы генерации	567
Empty	567
Range и Repeat	568
Глава 10. LINQ to XML	569
Обзор архитектуры	569
Что собой представляет DOM-модель	569
DOM-модель LINQ to XML	570
Обзор модели X-DOM	570
Загрузка и разбор	572
Сохранение и сериализация	573
Создание экземпляра X-DOM	573
Функциональное построение	574
Указание содержимого	575
Автоматическое глубокое копирование	576
Навигация и запросы	576
Навигация по дочерним узлам	576
Навигация по родительским узлам	580
Навигация по равноправным узлам	580
Навигация по атрибутам	581
Обновление модели X-DOM	581
Обновление простых значений	581
Обновление дочерних узлов и атрибутов	582
Обновление через родительский элемент	582

Работа со значениями	584
Установка значений	584
Получение значений	585
Значения и узлы со смешанным содержимым	586
Автоматическая конкатенация XText	586
Документы и объявления	587
XDocument	587
Объявления XML	589
Имена и пространства имен	590
Пространства имен в XML	591
Указание пространств имен в X-DOM	593
Модель X-DOM и стандартные пространства имен	594
Префиксы	595
Аннотации	596
Проектирование в модель X-DOM	597
Устранение пустых элементов	599
Потоковая передача проекции	600
Глава 11. Другие технологии XML и JSON	601
XmlReader	601
Чтение узлов	602
Чтение элементов	604
Чтение атрибутов	607
Пространства имен и префиксы	608
XmlWriter	609
Запись атрибутов	610
Запись других типов узлов	611
Пространства имен и префиксы	611
Шаблоны для использования XmlReader/XmlWriter	611
Работа с иерархическими данными	611
Смешивание XmlReader/XmlWriter с моделью X-DOM	615
Работа с JSON	616
Utf8JsonReader	617
Utf8JsonWriter	619
JsonDocument	620
Класс JsonNode	624
Глава 12. Освобождение и сборка мусора	631
IDisposable, Dispose и Close	631
Стандартная семантика освобождения	632
Когда выполнять освобождение	633
Очистка полей при освобождении	635
Анонимное освобождение	636
Автоматическая сборка мусора	637
Корневые объекты	639

Финализаторы	639
Вызов метода <code>Dispose</code> из финализатора	641
Восстановление	642
Как работает сборщик мусора	644
Приемы оптимизации	645
Принудительный запуск сборки мусора	649
Настройка сборки мусора во время выполнения	650
Нагрузка на память	650
Организация пула массивов	650
Утечки управляемой памяти	651
Таймеры	653
Диагностика утечек памяти	654
Слабые ссылки	654
Слабые ссылки и кеширование	656
Слабые ссылки и события	656
Глава 13. Диагностика	659
Условная компиляция	659
Сравнение условной компиляции и статических переменных-флагов	660
Атрибут <code>Conditional</code>	661
Классы <code>Debug</code> и <code>Trace</code>	663
<code>Fail</code> и <code>Assert</code>	663
<code>TraceListener</code>	664
Сброс и закрытие прослушивателей	666
Интеграция с отладчиком	666
Присоединение и останов	666
Атрибуты отладчика	667
Процессы и потоки процессов	667
Исследование выполняющихся процессов	667
Исследование потоков в процессе	668
StackTrace и StackFrame	668
Журналы событий Windows	670
Запись в журнал событий	671
Чтение журнала событий	672
Мониторинг журнала событий	672
Счетчики производительности	673
Перечисление доступных счетчиков производительности	673
Чтение данных счетчика производительности	675
Создание счетчиков и запись данных о производительности	676
Класс <code>Stopwatch</code>	677
Межплатформенные инструменты диагностики	678
<code>dotnet-counters</code>	678
<code>dotnet-trace</code>	679
<code>dotnet-dump</code>	681

Глава 14. Параллелизм и асинхронность	683
Введение	683
Многопоточная обработка	684
Создание потока	684
Join и Sleep	686
Блокирование	687
Локальное или совместно используемое состояние	688
Блокировка и безопасность потоков	690
Передача данных потоку	691
Обработка исключений	693
Потоки переднего плана или фоновые потоки	694
Приоритет потока	695
Передача сигналов	696
Многопоточность в обогащенных клиентских приложениях	696
Контексты синхронизации	698
Пул потоков	699
Задачи	701
Запуск задачи	702
Возвращение значений	704
Исключения	704
Продолжение	705
TaskCompletionSource	707
Task.Delay	710
Принципы асинхронности	710
Сравнение синхронных и асинхронных операций	710
Что собой представляет асинхронное программирование	711
Асинхронное программирование и продолжение	712
Важность языковой поддержки	714
Асинхронные функции в C#	716
Ожидание	716
Написание асинхронных функций	722
Асинхронные лямбда-выражения	727
Асинхронные потоки данных	728
Асинхронные методы в WinRT	730
Асинхронность и контексты синхронизации	731
Оптимизация	732
Асинхронные шаблоны	736
Отмена	736
Сообщение о ходе работ	738
Асинхронный шаблон, основанный на задачах	740
Комбинаторы задач	741
Асинхронное блокирование	745
Устаревшие шаблоны	745
Модель асинхронного программирования	745
Асинхронный шаблон на основе событий	746
BackgroundWorker	747

Глава 15. Потоки данных и ввод-вывод	749
Потоковая архитектура	749
Использование потоков	751
Чтение и запись	753
Поиск	754
Закрытие и сбрасывание	755
Тайм-ауты	755
Безопасность в отношении потоков управления	755
Потоки с опорными хранилищами	756
FileStream	756
MemoryStream	760
PipeStream	760
BufferedStream	765
Адаптеры потоков	765
Текстовые адаптеры	766
Двоичные адаптеры	771
Закрытие и освобождение адаптеров потоков	772
Потоки со сжатием	773
Сжатие в памяти	775
Сжатие файлов с помощью gzip в Unix	775
Работа с ZIP-файлами	776
Работа с файлами Tar	777
Операции с файлами и каталогами	778
Класс File	779
Класс Directory	783
FileInfo и DirectoryInfo	783
Path	784
Специальные папки	786
Запрашивание информации о томе	787
Перехват событий файловой системы	788
Безопасность, обеспечивающая операционной системой	789
Выполнение под учетной записью стандартного пользователя	790
Повышение административных полномочий и виртуализация	791
Размещенные в памяти файлы	792
Размещенные в памяти файлы и произвольный файловый ввод-вывод	792
Размещенные в памяти файлы и совместно используемая память (Windows)	793
Межплатформенная память, совместно используемая процессами	794
Работа с аксессорами представлений	794
Глава 16. Взаимодействие с сетью	797
Сетевая архитектура	797
Адреса и порты	800
Идентификаторы URI	801
HttpClient	803
Прокси-серверы	807
Аутентификация	808

Аутентификация через заголовки	809
Заголовки	810
Строки запросов	810
Выгрузка данных формы	811
Cookie-наборы	811
Реализация HTTP-сервера	812
Использование DNS	815
Отправка сообщений электронной почты с помощью SmtpClient	816
Использование TCP	817
Параллелизм и TCP	819
Получение почты POP3 с помощью TCP	820
Глава 17. Сборки	823
Содержимое сборки	823
Манифест сборки	824
Манифест приложения (Windows)	825
Модули	826
Класс Assembly	826
Строгие имена и подписание сборок	828
Назначение сборке строгого имени	828
Имена сборок	829
Полностью заданные имена	829
Класс AssemblyName	830
Информационная и файловая версии сборки	831
Подпись Authenticode	831
Подписание с помощью системы Authenticode	832
Ресурсы и подчиненные сборки	834
Встраивание ресурсов напрямую	835
Файлы .resources	836
Файлы .resx	837
Подчиненные сборки	839
Культуры и подкультуры	841
Загрузка, распознавание и изолирование сборок	842
Контексты загрузки сборок	844
Стандартный контекст ALC	849
“Текущий” контекст ALC	851
Метод Assembly.Load и контекстные ALC	851
Загрузка и распознавание неуправляемых библиотек	854
Класс AssemblyDependencyResolver	855
Выгрузка контекстов ALC	856
Унаследованные методы загрузки	857
Реализация системы подключаемых модулей	858
Глава 18. Рефлексия и метаданные	865
Рефлексия и активизация типов	866
Получение экземпляра Type	866
Имена типов	868

Базовые типы и интерфейсы	869
Создание экземпляров типов	870
Обобщенные типы	872
Рефлексия и вызов членов	873
Типы членов	875
Сравнение членов C# и членов CLR	877
Члены обобщенных типов	879
Динамический вызов члена	879
Параметры методов	880
Использование делегатов для повышения производительности	882
Доступ к неоткрытым членам	883
Обобщенные методы	884
Анонимный вызов членов обобщенного интерфейса	884
Вызов статических виртуальных/абстрактных членов интерфейсов	887
Рефлексия сборок	888
Модули	889
Работа с атрибутами	889
Основы атрибутов	889
Атрибут AttributeUsage	891
Определение собственного атрибута	892
Извлечение атрибутов во время выполнения	893
Динамическая генерация кода	894
Генерация кода IL с помощью класса DynamicMethod	895
Стек вычислений	896
Передача аргументов динамическому методу	897
Генерация локальных переменных	898
Ветвление	899
Создание объектов и вызов методов экземпляра	899
Обработка исключений	901
Выпуск сборок и типов	902
Объектная модель Reflection.Emit	903
Выпуск членов типа	904
Выпуск методов	905
Выпуск полей и свойств	907
Выпуск конструкторов	908
Присоединение атрибутов	909
Выпуск обобщенных методов и типов	910
Определение обобщенных методов	910
Определение обобщенных типов	912
Сложности, связанные с генерацией	912
Несозданные закрытые обобщения	912
Циклические зависимости	913
Синтаксический разбор IL	915
Написание дизассемблера	916

Глава 19. Динамическое программирование	921
Исполняющая среда динамического языка	921
Динамическое распознавание перегруженных членов	923
Упрощение паттерна “Посетитель”	923
Анонимный вызов членов обобщенного типа	926
Реализация динамических объектов	930
DynamicObject	930
ExpandoObject	932
Взаимодействие с динамическими языками	933
Передача состояния между C# и сценарием	934
Глава 20. Криптография	935
Обзор	935
Защита данных Windows	935
Хеширование	937
Алгоритмы хеширования в .NET	938
Хеширование паролей	939
Симметричное шифрование	939
Шифрование в памяти	941
Соединение в цепочку потоков шифрования	943
Освобождение объектов шифрования	944
Управление ключами	944
Шифрование с открытым ключом и подписание	945
Класс RSA	946
Цифровые подписи	947
Глава 21. Расширенная многопоточность	949
Обзор синхронизации	950
Монопольное блокирование	950
Оператор lock	951
Monitor.Enter и Monitor.Exit	952
Выбор объекта синхронизации	953
Когда нужна блокировка	953
Блокирование и атомарность	955
Вложенное блокирование	955
Взаимоблокировки	956
Производительность	957
Mutex	958
Блокирование и безопасность к потокам	959
Безопасность к потокам и типы .NET	960
Безопасность к потокам в серверах приложений	962
Неизменяемые объекты	964
Немонопольное блокирование	965
Семафор	965
Блокировки объектов чтения/записи	968

Сигнализирование с помощью дескрипторов ожидания событий	973
AutoResetEvent	973
ManualResetEvent	977
CountdownEvent	978
Создание межпроцессного объекта EventWaitHandle	978
Дескрипторы ожидания и продолжение	979
WaitAny, WaitAll и SignalAndWait	980
Класс Barrier	981
Ленивая инициализация	982
Lazy<T>	983
LazyInitializer	984
Локальное хранилище потока	985
[ThreadStatic]	986
ThreadLocal<T>	986
GetData и SetData	987
AsyncLocal<T>	987
Таймеры	989
PeriodicTimer	989
Многопоточные таймеры	990
Однопоточные таймеры	992
Глава 22. Параллельное программирование	993
Для чего нужна инфраструктура PFX?	994
Концепции PFX	994
Компоненты PFX	995
Когда необходимо использовать инфраструктуру PFX?	996
PLINQ	997
Продвижение параллельного выполнения	999
PLINQ и упорядочивание	1000
Ограничения PLINQ	1001
Пример: параллельная программа проверки орфографии	1001
Функциональная чистота	1004
Установка степени параллелизма	1004
Отмена	1005
Оптимизация PLINQ	1006
Класс Parallel	1011
Parallel.Invoke	1012
Parallel.For и Parallel.ForEach	1013
Параллелизм задач	1018
Создание и запуск задач	1019
Ожидание на множестве задач	1021
Отмена задач	1021
Продолжение	1022
Планировщики задач	1027
TaskFactory	1027

Работа с AggregateException	1028
Flatten и Handle	1029
Параллельные коллекции	1031
IProducerConsumerCollection<T>	1032
ConcurrentBag<T>	1033
BlockingCollection<T>	1034
Реализация очереди производителей/потребителей	1035
Глава 23. Span<T> и Memory<T>	1039
Промежутки и нарезание	1040
CopyTo и TryCopyTo	1042
Поиск в промежутках	1043
Работа с текстом	1043
Memory<T>	1044
Однонаправленные перечислители	1046
Работа с выделяемой в стеке и неуправляемой памятью	1048
Глава 24. Способность к взаимодействию	1051
Обращение к низкоуровневым DLL-библиотекам	1051
Маршализация типов и параметров	1052
Маршализация общих типов	1052
Маршализация классов и структур	1054
Маршализация параметров <i>in</i> и <i>out</i>	1056
Соглашения о вызовах	1056
Обратные вызовы из неуправляемого кода	1057
Обратные вызовы с помощью указателей на функции	1057
Обратные вызовы с помощью делегатов	1059
Эмуляция объединения C	1060
Совместно используемая память	1061
Отображение структуры на неуправляемую память	1063
fixed и fixed { . . . }	1066
Взаимодействие с COM	1068
Назначение COM	1068
Основы системы типов COM	1068
Обращение к компоненту COM из C#	1070
Необязательные параметры и именованные аргументы	1071
Неявные параметры <i>ref</i>	1071
Индексаторы	1072
Динамическое связывание	1072
Внедрение типов взаимодействия	1073
Эквивалентность типов	1074
Открытие объектов C# для COM	1074
Включение COM без регистрации	1076

Глава 25. Регулярные выражения	1077
Основы регулярных выражений	1077
Скомпилированные регулярные выражения	1079
Перечисление флагов <code>RegexOptions</code>	1079
Отмена символов	1081
Наборы символов	1081
Квантификаторы	1082
Жадные или ленивые квантификаторы	1083
Утверждения нулевой ширины	1084
Просмотр вперед и просмотр назад	1084
Привязки	1085
Границы слов	1086
Группы	1087
Именованные группы	1087
Замена и разделение текста	1088
Делегат <code>MatchEvaluator</code>	1089
Разделение текста	1090
Рецептурный справочник по регулярным выражениям	1090
Рецепты	1090
Справочник по языку регулярных выражений	1093
Предметный указатель	1098

Об авторе

Джозеф Албахари — автор книг *C# 10 in a Nutshell* и *LINQ Pocket Reference*. Он также является создателем LINQPad — популярной утилиты для подготовки кода и проверки запросов LINQ.

Об иллюстрации на обложке

Животное, изображенное на обложке книги — это журавль-красавка (лат. *Grus virgo*), называемый так за свою грацию и гармоничность. Данный вид журавля считается местным для Европы и Азии; на зимний период его представители мигрируют в Индию, Пакистан и северо-восточную Африку.

Хотя журавли-красавки являются самыми маленькими среди семейства журавлиных, они защищают свои территории так же агрессивно, как и другие виды журавлей, громкими голосами предупреждая других особей о нарушении границы и при необходимости вступая в бой. Журавли-красавки гнездятся не в болотистой местности, а на возвышенностях. Они могут жить даже в пустынях при наличии воды на расстоянии от 200 до 500 м. Временами для кладки яиц журавли-красавки строят гнезда, окружая их мелкими камешками, но чаще откладывают яйца прямо на землю, довольствуясь защитой только со стороны растительности.

В некоторых странах журавли-красавки считаются символом удачи и иногда защищены законом. Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой уничтожения; все они важны для нашего мира.

Изображение на обложке основано на черно-белой гравюре из книги *Wood's Illustrated Natural History*.

Предисловие

Язык C# 12 представляет собой девятое крупное обновление флагманского языка программирования от Microsoft, позиционирующее C# как язык с невероятной гибкостью и широтой применения. С одной стороны, он предлагает высокоуровневые абстракции, подобные выражениям запросов и асинхронным продолжениям, а с другой стороны, обеспечивает низкоуровневую эффективность через такие конструкции, как специальные типы значений и необязательные указатели.

Платой за развитие становится относительно трудное освоение языка. Несмотря на то что инструменты вроде Microsoft IntelliSense (и онлайновых справочников) великолепно помогают в работе, они предполагают наличие концептуальных знаний. Настоящая книга предлагает такие знания в сжатой и унифицированной форме, не утомляя беспорядочными и длинными вступлениями.

Подобно предшествующим семи изданиям книга организована вокруг концепций и сценариев использования, что делает ее пригодной как для последовательного чтения, так и для просмотра в произвольном порядке. Хотя допускается наличие только базовых навыков, материал анализируется довольно глубоко, что делает книгу ориентированной на читателей средней и высокой квалификации.

В книге рассматриваются язык C#, общязыковая исполняющая среда (Common Language Runtime — CLR) и библиотека базовых классов .NET 8 (Base Class Library — BCL). Такой подход был выбран для того, чтобы оставить место для раскрытия сложных и обширных тем без ущерба в отношении глубины или читабельности. Функциональные возможности, недавно добавленные в C#, специально помечаются, поэтому настоящую книгу можно применять также в качестве справочника по версиям C# 11 и C# 10.

Предполагаемая читательская аудитория

Книга рассчитана на читателей средней и высокой квалификации. Предварительное знание языка C# не обязательно, но необходимо наличие общего опыта программирования. Для начинающих данная книга будет дополнять, но не заменять вводный учебник по программированию.

Эта книга является идеальным дополнением к любой из огромного множества книг, ориентированных на прикладные технологии, такие как ASP.NET Core или Windows Presentation Foundation (WPF). В данной книге подробно рассматриваются язык и платформа .NET, чему обычно в книгах, посвященных прикладным технологиям, уделяется мало внимания (и наоборот).

Если вы ищете книгу, в которой кратко описаны все технологии .NET, то данная книга не для вас. Она также не подойдет, если вы стремитесь изучить API-интерфейсы, применяемые при разработке приложений для мобильных устройств.

Как организована эта книга

В главах 2–4 внимание сосредоточено целиком на языке C#, начиная с описания основ синтаксиса, типов и переменных и заканчивая такими сложными темами, как небезопасный код и директивы препроцессора. Новички должны читать указанные главы последовательно.

Остальные главы посвящены библиотеке базовых классов .NET 8. В них раскрываются такие темы, как LINQ, XML, коллекции, параллелизм, ввод-вывод и работа в сети, управление памятью, рефлексия, динамическое программирование, атрибуты, криптография и собственная способность к взаимодействию. Большинство этих глав можно читать в произвольном порядке кроме глав 5 и 6, где закладывается фундамент для последующих тем. Три главы, посвященные LINQ, тоже лучше читать последовательно, а в некоторых главах предполагается наличие общих знаний параллелизма, который раскрывается в главе 14.

Что требуется для работы с этой книгой

Примеры, приводимые в книге, требуют наличия .NET 8. Также полезно иметь под рукой документацию по .NET от Microsoft, чтобы просматривать справочную информацию об отдельных типах и членах (документация доступна по ссылке <https://learn.microsoft.com/en-us/dotnet/>).

Хотя исходный код можно писать в простом текстовом редакторе и компилировать программу в командной строке, намного продуктивнее работать с инструментом подготовки кода для немедленного тестирования фрагментов кода и с интегрированной средой разработки (Integrated Development Environment — IDE) для построения исполняемых файлов и библиотек.

В качестве инструмента подготовки кода для Windows загрузите LINQPad 8 из веб-сайта www.linqpad.net (он бесплатен). Инструмент LINQPad полностью поддерживает C# 12 и сопровождается автором книги.

В качестве IDE-среды для Windows загрузите Visual Studio 2022: подойдет любая редакция. В качестве межплатформенной IDE-среды загрузите Visual Studio Code.



Все листинги с кодом для всех глав доступны в виде интерактивных (редактируемых) примеров LINQPad. Для загрузки примеров перейдите на вкладку Samples (Примеры) в окне LINQPad, щелкните на ссылке Download/import more samples (Загрузить/импортировать дополнительные примеры) и затем в открывшемся окне щелкните на ссылке Download C# 12 in a Nutshell samples (Загрузить примеры C# 12 in a Nutshell).

Соглашения, используемые в этой книге

Для иллюстрации отношений между типами в книге применяется базовая система обозначений UML (рис. 0.1). Параллелограммом обозначается абстрактный класс, а окружностью — интерфейс. С помощью линии с незакрашенным треугольником обозначается наследование, причем треугольник указывает на базовый тип. Линия со стрелкой определяет одностороннюю ассоциацию, а линия без стрелки — двунаправленную ассоциацию.

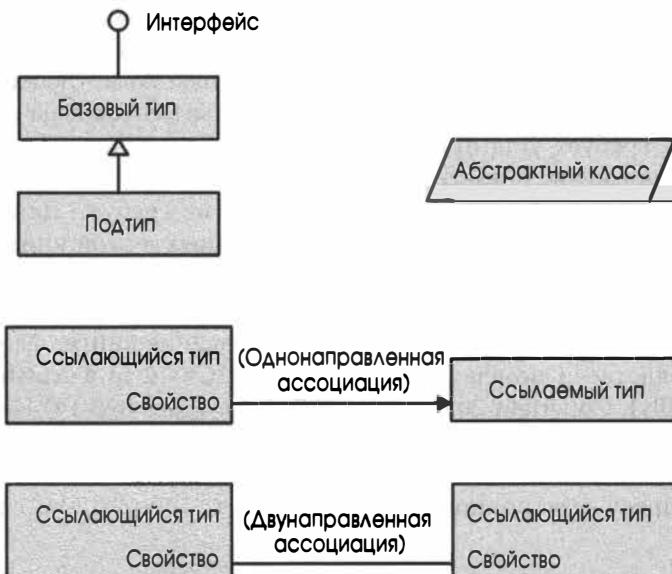


Рис. 0.1. Пример диаграммы

В книге также приняты следующие типографские соглашения.

Курсив

Применяется для новых терминов.

Моноширинный

Применяется для кода C#, ключевых слов и идентификаторов, а также вывода из программ.

Моноширинный полужирный

Применяется для выделения части кода.

Моноширинный курсив

Применяется для выделения текста, который должен быть заменен значениями, предоставленными пользователем.



Здесь приводится совет, указание или общее замечание.



Здесь приводится предупреждение или предостережение.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т.д.) доступны для загрузки по ссылке <http://www.albahari.com/nutshell>.

Настоящая книга призвана помочь вам выполнять свою работу. Обычно если в книге предлагается пример кода, то вы можете применять его в собственных программах и документации. Вы не обязаны обращаться к нам за разрешением, если только не используете значительную долю кода. Скажем, написание программы, в которой задействовано несколько фрагментов кода из этой книги, разрешения не требует. Для продажи или распространения кода примеров из книг O'Reilly разрешение обязательно. Ответ на вопрос путем цитирования данной книги и ссылки на пример кода разрешения не требует. Для встраивания значительного объема кода примеров, рассмотренных в этой книге, в документацию по вашему продукту разрешение обязательно.

Мы высоко ценим указание авторства, хотя в общем случае не требуем этого. Установление авторства обычно включает название книги, фамилию и имя автора, издательство и номер ISBN. Например: "C# 12 in a Nutshell by Joseph Albahari (O'Reilly). Copyright 2024, Joseph Albahari, 978-1-098-14744-0".

Если вам кажется, что способ использования вами примеров кода выходит за законные рамки или упомянутые выше разрешения, тогда свяжитесь с нами по следующему адресу электронной почты: permissions@oreilly.com.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info.dialektika@gmail.com

WWW: <http://www.dialektika.com>

Благодарности

Джозеф Албахари

С момента своего первого издания в 2007 году эта книга опиралась на мнения превосходных технических рецензентов. За вклад в последние издания я хотел бы выразить особую благодарность Стивену Таубу, Пауло Моргадо, Фреду Силбербергу, Витеку Карасу, Аарону Робинсону, Яну Ворличеку, Сэмю Джентайлу, Роду Стивенсу, Джареду Парсонсу, Метью Груву, Диксину Яну, Ли Коварду, Бонни Девитту, Вонсоку Чхэ, Лори Лалонду и Джеймсу Монтеманью.

За вклад в предыдущие издания я благодарю Эрика Липперта, Джона Скита, Стивена Тауба, Николаса Палдино, Криса Барроуза, Шона Фаркаса, Брайана Грёнкмейера, Маони Стивенс, Девида Де Винтера, Майка Барнетта, Мелитту Андерсен, Митча Вита, Брайана Пика, Кшиштофа Цвалина, Мэтта Уоррена, Джоэля Побара, Глин Гриффитс, Иона Василиана, Брэда Абрамса, Сэма Джинтайла и Адама Натана.

Я высоко ценю тот факт, что многие технические рецензенты являются со-стоявшимися личностями в компании Microsoft, и особенно благодарен им за то, что они уделили время, способствуя переходу настоящей книги на новый качественный уровень.

Хочу поблагодарить Бена Албахари и Эрика Иогансена, которые внесли свой вклад в предыдущие издания, а также команду O'Reilly — в особенности моего эффективного и отзывчивого редактора Корбина Коллинза. Наконец, я глубоко признателен моей замечательной жене Ли Албахари, чье присутствие делало меня счастливым на протяжении всего проекта.



Введение в C# и .NET

C# является универсальным, безопасным в отношении типов, объектно-ориентированным языком программирования. Цель C# заключается в обеспечении продуктивности работы программистов. Для этого в языке соблюдается баланс между простотой, выразительностью и производительностью. С самой первой версии главным архитектором языка C# был Андерс Хейлсберг (создатель Turbo Pascal и архитектор Delphi). Язык C# нейтрален в отношении платформ и работает с рядом исполняющих сред, специфичных для платформ.

Объектная ориентация

В языке C# предлагается расширенная реализация парадигмы объектной ориентации, которая включает *инкапсуляцию*, *наследование* и *полиморфизм*. Инкапсуляция означает создание вокруг *объекта* границы, предназначеннной для отделения внешнего (открытого) поведения от внутренних (закрытых) деталей реализации. Ниже перечислены отличительные особенности языка C# с объектно-ориентированной точки зрения.

- **Унифицированная система типов.** Фундаментальным строительным блоком в C# является инкапсулированная единица данных и функций, которая называется *типов*. Язык C# имеет унифицированную систему типов, где все типы в конечном итоге разделяют общий базовый тип. Это значит, что все типы, независимо от того, представляют они бизнес-объекты или примитивные сущности вроде чисел, совместно используют одну и ту же базовую функциональность. Например, экземпляр любого типа может быть преобразован в строку вызовом его метода *ToString*.
- **Классы и интерфейсы.** В рамках традиционной объектно-ориентированной парадигмы единственной разновидностью типа считается класс. В языке C# присутствуют типы других видов, одним из которых является *интерфейс*. Интерфейс похож на класс, не позволяющий хранить данные. Это означает, что он способен определять только *поведение* (но не *состояние*), что делает возможным множественное наследование, а также отделение спецификации от реализации.

- **Свойства, методы и события.** В чистой объектно-ориентированной парадигме все функции представляют собой *методы*. В языке C# методы являются только одной разновидностью функций-членов, куда также относятся *свойства* и *события* (помимо прочего). Свойства — это функции-члены, которые инкапсулируют фрагмент состояния объекта, такой как цвет кнопки или текст метки. События — это функции-члены, упрощающие выполнение действий при изменении состояния объекта.

Хотя C# — главным образом объектно-ориентированный язык, он также кое-что заимствует из парадигмы *функционального программирования*. Конкретные заимствования перечислены ниже.

- **Функции могут трактоваться как значения.** За счет применения *делегатов* язык C# позволяет передавать функции как значения в другие функции и получать их из других функций.
- **Для чистоты в C# поддерживаются шаблоны.** Основная черта функционального программирования заключается в том, чтобы избегать использования переменных, значения которых изменяются, отдавая предпочтение декларативным шаблонам. В языке C# имеются ключевые средства, содействующие таким шаблонам, в том числе возможность написания на лету неименованных функций, которые “захватывают” переменные (*лямбда-выражения*), и возможность спискового или реактивного программирования через *выражения запросов*. В C# также предоставляются записи, которые облегчают написание *неизменяемых* (допускающих только чтение) типов.

Безопасность в отношении типов

Прежде всего, C# является языком, *безопасным к типам*. Это означает, что экземпляры типов могут взаимодействовать только через определяемые ими протоколы, обеспечивая тем самым внутреннюю согласованность каждого типа. Например, C# не позволяет взаимодействовать со *строковым* типом так, как если бы он был *целочисленным* типом.

Говоря точнее, в C# поддерживается *статическая типизация*, при которой язык обеспечивает безопасность к типам *на этапе компиляции*, дополняя ею безопасность в отношении типов, навязываемую *во время выполнения*.

Статическая типизация ликвидирует обширную категорию ошибок еще до запуска программы. Она перекладывает бремя проверки того, что все типы в программе корректно подходят друг другу, с модульных тестов времени выполнения на компилятор. В результате крупные программы становятся намного более легкими в управлении, более предсказуемыми и более надежными. Кроме того, статическая типизация позволяет таким инструментам, как *IntelliSense* в *Visual Studio*, оказывать помощь в написании программы: поскольку тип заданной переменной известен, то известны и методы, которые можно вызывать с этой переменной. Инструменты подобного рода способны также повсюду в программе распознавать, что переменная, тип или метод используется, делая возможным надежный рефакторинг.



Язык C# также позволяет частям кода быть динамически типизированными через ключевое слово `dynamic`. Тем не менее, C# остается преимущественно статически типизированным языком.

Язык C# также называют *строго типизированным*, потому что его правила, касающиеся типов (применяемые как статически, так и во время выполнения), являются очень строгими. Например, невозможно вызвать функцию, которая предназначена для приема целого числа, с числом с плавающей точкой, не выполнив предварительно явное преобразование числа с плавающей точкой в целое. Это помогает предотвращать ошибки.

Управление памятью

В плане реализации автоматического управления памятью C# полагается на исполняющую среду. Общязыковая исполняющая среда (Common Language Runtime — CLR) имеет сборщик мусора, выполняющийся как часть вашей программы; он восстанавливает память, занятую объектами, на которые больше нет ссылок. В итоге с программистов снимается обязанность по явному освобождению памяти для объекта, что устраняет проблему некорректных указателей, которая встречается в языках вроде C++.

Язык C# не исключает указатели: он просто делает их необязательными при решении большинства задач программирования. Для “горячих” точек, критичных к производительности, и возможности взаимодействия указатели и явное выделение памяти разрешены в блоках, которые явно помечены как `unsafe` (т.е. небезопасные).

Поддержка платформ

В C# имеются исполняющие среды, которые поддерживают следующие платформы:

- Windows 7+ Desktop (для обогащенных клиентских приложений, веб-приложений, серверных приложений и приложений командной строки);
- macOS (для веб-приложений и приложений командной строки, а также обогащенных клиентских приложений через Mac Catalyst);
- Linux (для веб-приложений и приложений командной строки);
- Android и iOS (для мобильных приложений);
- устройства Windows 10 (Xbox, Surface Hub и HoloLens) через UWP.

Существует также технология под названием *Blazor*, позволяющая компилировать код C# в веб-сборку, которая может выполняться в браузере.

Общеязыковые исполняющие среды, библиотеки базовых классов и исполняющие среды

Поддержка времени выполнения для программ C# состоит из *общеязыковой исполняющей среды* и *библиотеки базовых классов*. Исполняющая среда может также включать *прикладной слой* более высокого уровня, который содержит библиотеки для разработки обогащенных клиентских, мобильных или веб-приложений (рис. 1.1). Доступны разные исполняющие среды, предназначенные для отличающихся видов приложений, а также различных платформ.

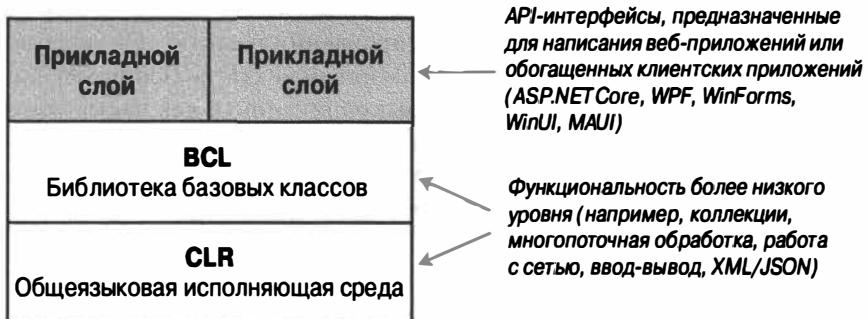


Рис. 1.1. Архитектура исполняющей среды

Общеязыковая исполняющая среда

Общеязыковая исполняющая среда (CLR) предоставляет важные службы времени выполнения, такие как автоматическое управление памятью и обработка исключений. (Понятие “общеязыковая” относится к тому факту, что та же самая среда может использоваться другими управляемыми языками вроде F#, Visual Basic и Managed C++.)

C# называют *управляемым языком*, потому что его компилятор компилирует исходный код в управляемый код, который представлен на *промежуточном языке* (Intermediate Language — IL). Среда CLR преобразует код IL в собственный код машины, такой как X64 или X86, обычно прямо перед выполнением. Такое преобразование называется *оперативной* (Just-In-Time — JIT) компиляцией. Доступна также *ранняя* (Ahead-Of-Time — AOT) компиляция для улучшения показателей времени начального запуска в случае крупных сборок или устройств с ограниченными ресурсами (а также для удовлетворения правил хранилища приложений iOS при разработке мобильных приложений).

Контейнер для управляемого кода называется *сборкой* (assembly). Сборка содержит не только код IL, но также информацию о типах (*метаданные*). Наличие метаданных дает возможность сборкам ссылаться на типы в других сборках, не нуждаясь в дополнительных файлах.



Вы можете исследовать и дизассемблировать содержимое сборки посредством инструмента `ildasm` от Microsoft. А такие инструменты, как `ILSpy` или `dotPeek` от JetBrains, позволяют продвинуться еще дальше и декомпилировать код IL в C#. Поскольку по сравнению с собственным машинным кодом код IL является более высокоуровневым, декомпилятор способен проделать неплохую работу по воссозданию исходного кода C#.

Программа может запрашивать собственные метаданные (*рефлексия*) и даже генерировать новый код IL во время выполнения (`Reflection.Emit`).

Библиотека базовых классов

Среда CLR всегда поставляется с комплектом сборок, который называется **библиотекой базовых классов** (Base Class Library — BCL). Библиотека BCL предлагает программистам основную функциональность, такую как коллекции, ввод-вывод, обработка текста, поддержка XML/JSON, работа с сетью, шифрование, возможность взаимодействия и параллельное программирование. В библиотеке BCL также реализованы типы, которые требуются самому языку C# (для средств, подобных перечислению, запрашиванию и асинхронности) и позволяют явно получать доступ к средствам CLR вроде рефлексии и управления памятью.

Исполняющие среды

Исполняющая среда (также называемая *инфраструктурой*) — это развертываемая единица, которую вы можете загрузить и установить. Исполняющая среда состоит из среды CLR (со своей библиотекой BCL) плюс необязательного **прикладного слоя**, специфичного для того типа приложения, которое вы пишете — веб-приложение, мобильное приложение, обогащенное клиентское приложение и т.д. (При написании консольного приложения командной строки или библиотеки, не предназначенной для пользовательского интерфейса, прикладной слой не нужен.)

Когда вы пишете приложение, то *нацеливаетесь* на конкретную исполняющую среду, т.е. ваше приложение использует и зависит от функциональности, которую обеспечивает исполняющая среда. Выбор исполняющей среды также определяет, какие платформы будет поддерживать приложение.

В следующей таблице перечислены основные варианты исполняющих сред.

Прикладной слой	CLR/BCL	Типы программ	Среды выполнения
ASP.NET	.NET 8	Веб-приложение	Windows, Linux, macOS
Windows Desktop	.NET 8	Приложение для Windows	Windows 10+
WinUI 3	.NET 8	Приложение для Windows	Windows 10+
MAUI	.NET 8	Мобильное приложение, настольное приложение	iOS, Android, macOS, Windows 10+
.NET Framework	.NET Framework	Веб-приложение, приложение для Windows	Windows 7+

На рис. 1.2 приведена графическая иллюстрация изложенных сведений, которая также служит путеводителем по темам, раскрываемым в настоящей книге.

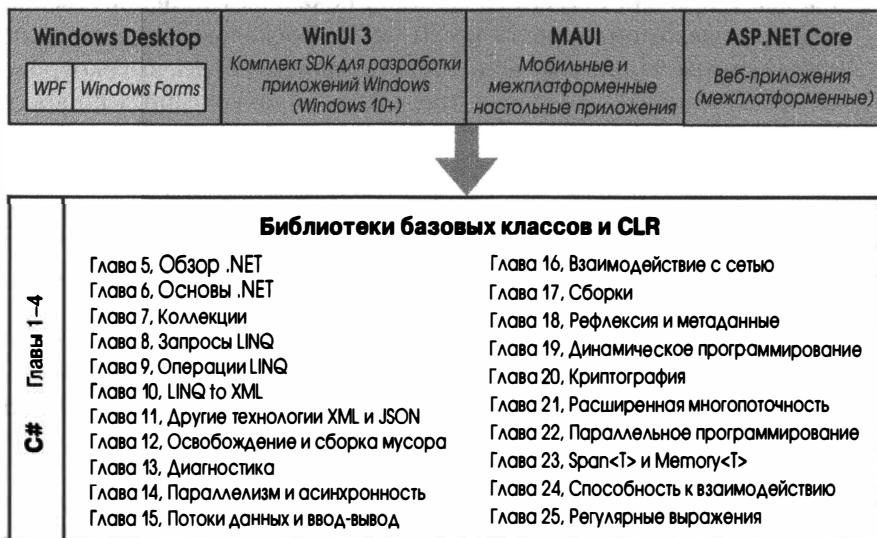


Рис. 1.2. Исполняющие среды для C#

.NET 8

.NET 8 — это флагманская исполняющая среда с открытым кодом производства Microsoft. Вы можете создавать веб-приложения и консольные приложения, которые выполняются под управлением Windows, Linux и macOS, обогащенные клиентские приложения, выполняемые под управлением Windows 10+ и macOS, а также мобильные приложения, которые выполняются под управлением iOS и Android. Основное внимание в книге уделяется CLR и BCL из .NET 8.

В отличие от .NET Framework исполняющая среда .NET 8 на машинах Windows заранее не устанавливается. Если вы попытаетесь запустить приложение .NET 8 в отсутствие корректной исполняющей среды, тогда отобразится сообщение, направляющее на веб-страницу, где можно загрузить исполняющую среду. Вы можете исключить попадание в такую ситуацию, создав *автономное* развертывание, которое включает части исполняющей среды, необходимые приложению.



Хронология обновлений .NET выглядит следующим образом: .NET Core 1.x → .NET Core 2.x → .NET Core 3.x → .NET 5 → .NET 6 → .NET 7 → .NET 8. После выхода .NET Core 3 в Microsoft решили удалить слово “Core” из названия и пропустить версию 4, чтобы избежать путаницы со средой .NET Framework 4.x, которая предшествует всем предыдущим исполняющим средам, но все еще поддерживается и широко используется. Это означает, что сборки, которые компилировались в версиях от .NET Core 1.x до .NET 7, в большинстве случаев будут функционировать без каких-либо изменений под управлением .NET 8. Напротив, сборки, скомпилированные в .NET Framework (любой версии), обычно несовместимы с .NET 8.

Windows Desktop и WinUI 3

При написании обогащенных клиентских приложений, которые выполняются под управлением Windows 10 и последующих версий, можно выбирать между классическими API-интерфейсами Windows Desktop (Windows Forms и WPF) и WinUI 3. API-интерфейсы Windows Desktop являются частью исполняющей среды .NET Desktop, а WinUI 3 — частью комплекта SDK для разработки приложений Windows (*Windows App SDK*), который должен загружаться отдельно.

Классические API-интерфейсы Windows Desktop существуют с 2006 года и пользуются великолепной поддержкой сторонними библиотеками, а также сопровождаются многочисленными ответами на вопросы на сайтах, подобных StackOverflow. Версия WinUI 3 была выпущена в 2022 году и предназначена для написания современных иммерсивных приложений с новейшими элементами управления Windows 10+. Она является преемником универсальной платформы Windows (*Universal Windows Platform* — UWP).

MAUI

Пользовательский интерфейс многоплатформенных приложений (Multi-platform App UI — MAUI) предназначен в первую очередь для создания мобильных приложений, работающих в средах iOS и Android, хотя его также можно применять для построения настольных приложений, функционирующих под управлением macOS и Windows через Mac Catalyst и WinUI 3. MAUI — это развитие Xamarin и позволяет нацеливать один проект на несколько платформ.



Для создания межплатформенных настольных приложений альтернативой MAUI является сторонняя библиотека по имени Avalonia, которая также запускается в среде Linux и архитектурно проще MAUI (поскольку работает без слоя косвенности Catalyst/WinUI). Avalonia имеет API-интерфейс, аналогичный WPF, и предлагает коммерческое дополнение под названием XPF, обеспечивающее почти полную совместимость с WPF.

.NET Framework

.NET Framework — это первоначальная исполняющая среда Microsoft, поддерживающая исключительно Windows, которая предназначена для написания веб-приложений и обогащенных клиентских приложений, запускаемых (только) на настольном компьютере или сервере Windows. Никаких крупных новых выпусков не планируется, хотя в Microsoft продолжат поддержку и сопровождение текущего выпуска 4.8 из-за обилия существующих приложений.

В .NET Framework среда CLR и библиотека BCL интегрированы с прикладным слоем. Приложения, написанные в .NET Framework, можно перекомпилировать в .NET 8, хотя обычно они требуют внесения ряда изменений. Некоторые средства .NET Framework отсутствуют в .NET 8 (и наоборот).

Исполняющая среда .NET Framework заранее установлена в Windows и автоматически обновляется через центр обновления Windows. В случае нацеливания на .NET Framework 4.8 вы можете использовать функциональные средства C# 7.3 и предшествующих версий. (Вы можете указать в файле проекта более

новую версию языка, что разблокирует все новейшие возможности языка, за исключением тех, которые требуют поддержки со стороны более новой среды выполнения.)



Слово “.NET” уже давно применяется в качестве широкого термина для обозначения любой технологии, которая включает “.NET” (.NET Framework, .NET Core, .NET Standard и т.д.).

Это означает, что переименование .NET Core в .NET создало досадную неоднозначность. Чтобы не возникало путаницы, новая платформа .NET в книге будет называться .NET 5+, а для ссылки на .NET Core и преемников будет использоваться фраза “.NET Core и .NET 5+”.

Усугубляя путаницу, .NET (5+) является инфраструктурой (framework), но она сильно отличается от .NET Framework. Таким образом, в книге по возможностям будет применяться термин *исполняющая среда*, а не *инфраструктура*.

Нишевые исполняющие среды

Следующие исполняющие среды все еще доступны:

- .NET Core 3.0 и 3.1 (заменена .NET 5);
- .NET Core 1.x и 2.x (только для веб-приложений и приложений командной строки);
- Windows Runtime для Windows 8/8.1 (теперь UWP);
- Microsoft XNA для разработки игр (теперь UWP);

Существуют также нишевые исполняющие среды, которые перечислены ниже.

- Unity является платформой для разработки игр, которая позволяет описывать логику игры с помощью C#.
- Универсальная платформа Windows (UWP) была спроектирована для написания сенсорных приложений, которые работают на настольных компьютерах и устройствах под управлением Windows 10+, включая Xbox, Surface Hub и HoloLens. Приложения UWP помещаются в песочницу и поставляются через Магазин Windows. UWP использует версию .NET Core 2.2 CLR/BCL, и маловероятно, что эта зависимость будет обновлена; замену в Microsoft рекомендуют пользователям перейти на современную замену UWP — WinUI 3. Но поскольку WinUI 3 поддерживает только рабочий стол Windows, с UWP по-прежнему связано нишевое применение для Xbox, Surface Hub и HoloLens.
- .NET Micro Framework предназначена для выполнения кода .NET на встроенных устройствах с крайне ограниченными ресурсами (менее одного мегабайта).

Возможно также выполнение управляемого кода внутри SQL Server. Благодаря интеграции CLR с SQL Server вы можете писать специальные функции, хранимые процедуры и агрегации на языке C# и затем вызывать их в коде SQL. Такой прием работает в сочетании с .NET Framework и специальной “размещенной” средой CLR, которая обеспечивает песочницу для защиты целостности процесса SQL Server.

Краткая история языка C#

Ниже приведена обратная хронология появления новых средств в каждой версии C#, которая будет полезна читателям, знакомым с более старыми версиями языка.

Нововведения версии C# 12

Версия C# 12 поставляется вместе с *Visual Studio 2022* и используется в случае нацеливания на .NET 8.

Выражения коллекций

Вместо того чтобы инициализировать массив следующим образом:

```
char[] vowels = {'a','e','i','o','u'};
```

теперь можно применять квадратные скобки (*выражение коллекции*):

```
char[] vowels = ['a','e','i','o','u'];
```

Выражения коллекций дают два основных преимущества. Во-первых, тот же самый синтаксис работает и с другими типами коллекций, такими как списки и наборы (и даже с низкоуровневыми типами диапазонов):

```
List<char> list      = ['a','e','i','o','u'];
HashSet<char> set      = ['a','e','i','o','u'];
ReadOnlySpan<char> span = ['a','e','i','o','u'];
```

Во-вторых, они имеют *целевую типизацию*, а это значит, что вы можете опустить тип в других сценариях, где компилятор способен его определить, например, при вызове методов:

```
Foo(['a','e','i','o','u']);
void Foo (char[] letters) { ... }
```

Дополнительные сведения ищите в разделе “Инициализаторы и выражения коллекций” в главе 4.

Основные конструкторы в классах и структурах

Начиная с версии C# 12, список параметров можно помещать непосредственно после объявления класса (или структуры):

```
class Person (string firstName, string lastName)
{
    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

Такой код инструктирует компилятор о необходимости автоматического создания основного конструктора, позволяющего выполнять следующее действие:

```
Person p = new Person ("Alice", "Jones");
p.Print(); // Alice Jones
```

Данное средство существует, начиная с введения записей в C# 9, где оно ведет себя несколько по-другому. В случае использования записей компилятор (по умолчанию) генерирует для каждого параметра основного конструктора открытое свойство, допускающее только инициализацию. К классам и структурам это не относится; для достижения того же результата свойства должны определяться явно:

```
class Person (string firstName, string lastName)
{
    public string FirstName { get; set; } = firstName;
    public string LastName { get; set; } = lastName;
}
```

Основные конструкторы хорошо работают в простых сценариях. Связанные с ними нюансы и ограничения будут описаны в разделе “Основные конструкторы (C# 12)” главы 3.

Параметры со стандартными значениями в лямбда-выражениях

Точно так же, как в обычных методах могут быть определены параметры со стандартными значениями:

```
void Print (string message = "") => Console.WriteLine (message);
```

их также можно определять в лямбда-выражениях:

```
var print = (string message = "") => Console.WriteLine (message);
print ("Hello");
print ();
```

Это средство полезно при работе с такими библиотеками, как ASP.NET Minimal API.

Снабжение псевдонимом любого типа

Язык C# всегда позволял снабжать псевдонимом простой или обобщенный тип с помощью директивы using:

```
using ListOfInt = System.Collections.Generic.List<int>;
var list = new ListOfInt();
```

Начиная с версии C# 12, такой подход работает и с типами других видов вроде массивов и кортежей:

```
using NumberList = double[];
using Point = (int X, int Y);

NumberList numbers = { 2.5, 3.5 };
Point p = (3, 4);
```

Прочие новые средства

В версии C# 12 также поддерживаются *встроенные массивы* через атрибут [System.Runtime.CompilerServices.InlineArray]. В результате появляется возможность создания массивов фиксированного размера в структуре без необходимости применения небезопасного контекста. Средство предназначено в первую очередь для использования в API-интерфейсах исполняющей среды.

Нововведения версии C# 11

Версия C# 11 поставляется вместе с *Visual Studio 2022* и используется в случае нацеливания на .NET 7.

Необработанные строковые литералы

Помещение строки в три или более символов кавычек приводит к созданию *необработанного строкового литерала*, который может содержать практически любую последовательность символов без необходимости в управляющих символах или удвоении. Это упрощает представление литералов JSON, XML и HTML, а также регулярных выражений и исходного кода:

```
string raw = """<file path="c:\temp\test.txt"></file>""";
```

Необработанные строковые литералы могут занимать несколько строчек и допускают интерполяцию через префикс \$:

```
string multiLineRaw = $$"
Line 1
Line 2
The date and time is {DateTime.Now}
""";
```

Использование двух (или большего количества) символов \$ в префикссе необработанной строки изменяет последовательность интерполяции с одной фигурной скобки на две и большее число фигурных скобок, что позволяет включать фигурные скобки в саму строку:

```
Console.WriteLine ($$""{ "TimeStamp": "{DateTime.Now}" }""");
// Вывод: { "TimeStamp": "01/01/2024 12:13:25 PM" }
```

Нюансы данного средства раскрываются в разделах “Необработанные строковые литералы (C# 11)” и “Интерполяция строк” главы 2.

Строки UTF-8

С помощью суффикса u8 создаются строковые литералы, закодированные в UTF-8, а не в UTF-16. Это средство предназначено для сложных сценариев, таких как низкоуровневая обработка текста JSON в горячих точках производительности:

```
ReadOnlySpan<byte> utf8 = "ab→cd"u8; // Символ стрелки занимает 3 байта
Console.WriteLine (utf8.Length); // 7
```

Лежащим в основе типом является *ReadOnlySpan<byte>* (глава 23), который можно преобразовать в массив байтов, вызвав его метод *ToArray()*.

Шаблоны списков

Шаблоны списков соответствуют последовательности элементов в квадратных скобках и работают с любыми типами коллекций, которые поддерживают подсчет (с помощью свойства Count или Length) и индексацию (посредством индексатора типа int или System.Index):

```
int[] numbers = { 0, 1, 2, 3, 4 };
Console.WriteLine (numbers is [0, 1, 2, 3, 4]);      // True
```

Подчеркивание соответствует одному элементу с любым значением, а две точки — нулю или большему количеству элементов (*срезу*):

```
Console.WriteLine (numbers is [_, 1, .., 4]);      // True
```

За срезом может следовать шаблон var — детали ищите в разделе “Шаблоны списков” главы 4.

Обязательные члены

Применение модификатора required к полю или свойству заставляет потребителей этого класса или структуры заполнять такой член через инициализатор объекта при его конструировании:

```
Asset a1 = new Asset { Name = "House" };      // Нормально
Asset a2 = new Asset();                        // Ошибка: не скомпилируется!
class Asset { public required string Name; }
```

Благодаря данному средству можно избежать написания конструкторов с длинными списками параметров, что способствует упрощению создания подклассов. Если вы также хотите написать конструктор, то можете применить атрибут [SetsRequiredMembers], чтобы обойти ограничение на обязательный член для этого конструктора — детали ищите в разделе “Обязательные члены (C# 11)” главы 3.

Статические виртуальные/абстрактные члены интерфейсов

Начиная с версии C# 11, в интерфейсах можно объявлять члены как static virtual (статические виртуальные) или static abstract (статические абстрактные):

```
public interface IParsable<TSelf>
{
    static abstract TSelf Parse (string s);
}
```

Такие члены реализованы в виде статических функций в классах или структурах и могут вызываться полиморфно через ограниченный параметр типа:

```
T ParseAny<T> (string s) where T : IParsable<T> => T.Parse (s);
```

Функции операций также можно объявлять как static virtual или static abstract. Дополнительные сведения представлены в разделе “Статические виртуальные/абстрактные члены интерфейсов” главы 3 и в разделе “Статический полиморфизм” главы 4. Кроме того, в разделе “Вызов статических виртуальных/абстрактных членов интерфейсов” главы 18 объясняется, как вызывать статические абстрактные члены через рефлексию.

Операция обобщенной математики

Интерфейс `System.Numerics.INumber<TSelf>` (появившийся в .NET 7) унифицирует арифметические операции для всех числовых типов, позволяя писать обобщенные методы следующего вида:

```
T Sum<T> (T[] numbers) where T : INumber<T>
{
    T total = T.Zero;
    foreach (T n in numbers)
        total += n;      // Вызывает операцию сложения для любого числового типа
    return total;
}

int intSum = Sum (3, 5, 7);
double doubleSum = Sum (3.2, 5.3, 7.1);
decimal decimalSum = Sum (3.2m, 5.3m, 7.1m);
```

`INumber<TSelf>` реализуется всеми вещественными и целочисленными числовыми типами в .NET (а также типом `char`) и включает в себя несколько интерфейсов, содержащих определения статических абстрактных операций, например:

```
static abstract TResult operator + (TSelf left, TOther right);
```

Данная тема будет раскрыта в разделах “Полиморфные операции” и “Обобщенная математика” главы 4.

Прочие новые средства

Доступ к типу с модификатором доступности `file` возможен только из того же самого файла, и он предназначен для использования в генераторах исходного кода:

```
file class Foo { ... }
```

В C# 11 также появились проверяемые операции (см. раздел “Проверяемые операции” главы 4) для определения функций операций, которые будут вызываться внутри блоков `checked` (это требовалось для полной реализации обобщенной математики). В C# 11 также было ослаблено требование к заполнению каждого поля в конструкторе структуры (см. раздел “Семантика конструирования структур” главы 3).

Наконец, целочисленные типы с собственным размером `nint` и `nuint`, которые были введены в C# 9 для соответствия адресному пространству процесса во время выполнения (32 или 64 бита), в версии C# 11 были улучшены в случае нацеливания на .NET 7 или последующую версию. В частности, различие на этапе компиляции между этими типами и их базовыми типами времени выполнения (`IntPtr` и `UIntPtr`) устранено при нацеливании на .NET 7+. Подробное обсуждение ищите в разделе “Целочисленные типы с собственным размером” главы 4.

Нововведения версии C# 10

Версия C# 10 поставляется вместе с *Visual Studio 2022* и используется в случае нацеливания на .NET 6.

Пространства имен с областью видимости на уровне файлов

В общем случае, когда все типы в файле определены в одном пространстве имен, объявление пространства имен с областью видимости на уровне файла в C# 10 уменьшает беспорядок и устраниет ненужный уровень отступов:

```
namespace MyNamespace; //Применяется ко всему, что находится далее в файле
class Class1 {}          // внутри MyNamespace
class Class2 {}          // внутри MyNamespace
```

Директива `global using`

В случае добавления к директиве `using` ключевого слова `global` директива применяется ко всем файлам в проекте:

```
global using System;
global using System.Collections.Generic;
```

Это позволяет избежать повторения одних и тех же директив в каждом файле. Директивы `global using` работают с `using static`.

Кроме того, проекты .NET 6 теперь поддерживают *неявные директивы global using*: если для элемента `ImplicitUsings` в файле проекта установлено значение `true`, тогда автоматически импортируются наиболее часто используемые пространства имен (в зависимости от типа проекта SDK). Более подробную информацию ищите в разделе “Директива `global using`” главы 2.

Неразрушающее изменение для анонимных типов

В C# 9 появилось ключевое слово `with` для выполнения неразрушающего изменения записей. В C# 10 ключевое слово `with` работает также с анонимными типами:

```
var a1 = new { A = 1, B = 2, C = 3, D = 4, E = 5 };
var a2 = a1 with { E = 10 };
Console.WriteLine (a2);      // { A = 1, B = 2, C = 3, D = 4, E = 10 }
```

Новый синтаксис деконструирования

В C# 7 был введен синтаксис деконструирования кортежей (или любого типа, имеющего метод `Deconstruct`). В C# 10 данный синтаксис развивает дальше, позволяя смешивать присваивание и объявление в одном действии деконструирования:

```
var point = (3, 4);
double x = 0;
(x, double y) = point;
```

Инициализаторы полей и конструкторы без параметров в структурах

Начиная с версии C# 10, в структуры можно включать инициализаторы полей и конструкторы без параметров (см. раздел “Структуры” в главе 3). Они выполняются только в случае явного вызова конструктора, поэтому их можно легко обойти, например, с помощью ключевого слова `default`. Данное средство было введено в первую очередь для нужд структур типа записей.

Структуры типа записей

Записи появились в версии C# 9, где они выступали в качестве расширяемого на этапе компиляции класса. В версии C# 10 записи также могут быть структурами:

```
record struct Point (int X, int Y);
```

В остальном правила похожи: *структуры типа записей* обладают почти такими же функциями, как и *структуры типа классов* (см. раздел “Записи” в главе 4). Исключением является то, что свойства структур типа записей, генерируемые компилятором, доступны для изменения, если только перед объявлением записи не указано ключевое слово `readonly`.

Усовершенствования в лямбда-выражениях

Синтаксис лямбда-выражений был усовершенствован в нескольких отношениях. Во-первых, разрешена неявная типизация (`var`):

```
var greeter = () => "Hello, world";
```

Неявным типом лямбда-выражения является делегат `Action` или `Func`, поэтому в данном случае `greeter` имеет тип `Func<string>`. Типы параметров должны быть заданы явно:

```
var square = (int x) => x * x;
```

Во-вторых, в лямбда-выражении можно указывать возвращаемый тип:

```
var sqr = int (int x) => x;
```

Это сделано в первую очередь для улучшения производительности компилятора при обработке сложных вложенных лямбда-выражений.

В-третьих, лямбда-выражение можно передавать в параметре метода типа `object`, `Delegate` или `Expression`:

```
M1 (() => "test"); // Неявно типизируется как Func<string>
M2 (() => "test"); // Неявно типизируется как Func<string>
M3 (() => "test"); // Неявно типизируется как Expression<Func<string>>

void M1 (object x) {}
void M2 (Delegate x) {}
void M3 (Expression x) {}
```

Наконец, к сгенерированному компилятором целевому методу лямбда-выражения (а также к его параметрам и возвращаемому значению) можно применять атрибуты:

```
Action a = [Description("test")] () => { };
```

За деталями обращайтесь в раздел “Применение атрибутов к лямбда-выражениям” главы 4.

Шаблоны вложенных свойств

В версии C# 10 допустим следующий упрощенный синтаксис для сопоставления с шаблонами вложенных свойств (см. раздел “Шаблоны свойств” в главе 4):

```
var obj = new Uri ("https://www.linqpad.net");
if (obj is Uri { Scheme.Length: 5 }) ...
```

А вот его эквивалент:

```
if (obj is Uri { Scheme: { Length: 5 } }) ...
```

Атрибут `CallerArgumentExpression`

Параметр метода, к которому применяется атрибут `[CallerArgumentExpression]`, захватывает выражение аргумента из места вызова:

```
Print (Math.PI * 2);
void Print (double number,
    [CallerArgumentExpression("number")] string expr = null)
    => Console.WriteLine (expr);
// Вывод: Math.PI * 2
```

Это средство предназначено в первую очередь для библиотек проверки и утверждений (см. раздел “Атрибут `CallerArgumentExpression`” в главе 4).

Остальные новые средства

Директива `#line` в версии C# 10 была усовершенствована и теперь позволяет указывать колонку и диапазон.

Интерполированные строки в C# 10 могут быть константами, если интерполированные значения являются константами.

Записи в версии C# 10 могут запечатывать метод `ToString()`.

Анализ определенного присваивания в C# был улучшен, и теперь работают выражения вроде показанного ниже:

```
if(foo?.TryParse ("123", out var number) ?? false)
    Console.WriteLine (number);
```

(До версии C# 10 компилятор выдавал ошибку “Use of unassigned local variable ‘number’” (Использование локальной переменной `number`, которой не было присвоено значение).)

Нововведения версии C# 9.0

Версия C# 9.0 поставляется вместе с *Visual Studio 2019* и используется в случае нацеливания на .NET 5.

Операторы верхнего уровня

С помощью *операторов верхнего уровня* (см. врезку “Операторы верхнего уровня” в главе 2) вы можете писать программу без метода `Main` и класса `Program`:

```
using System;
Console.WriteLine ("Hello, world");
```

Операторы верхнего уровня могут включать методы (которые действуют как локальные методы). Кроме того, можно получать доступ к аргументам командной строки через “магическую” переменную `args` и возвращать значение вызывающему компоненту. За операторами верхнего уровня могут следовать объявления типов и пространств имен.

Средства доступа только для инициализации

Средство доступа только для инициализации (см. раздел “Средства доступа только для инициализации” в главе 3) в объявлении свойства использует ключевое слово `init` вместо `set`:

```
class Foo { public int ID { get; init; } }
```

Такое свойство ведет себя подобно свойству только для чтения, но также может устанавливаться через инициализатор объектов:

```
var foo = new Foo { ID = 123 };
```

Это позволяет создавать неизменяемые (допускающие только чтение) типы, которые можно заполнять посредством инициализатора объекта, а не конструктора, и помогает избавляться от конструкторов, принимающих большое количество необязательных параметров. Кроме того, средства доступа только для инициализации делают возможным *неразрушающее изменение* в случае применения в записях.

Записи

Запись (см. раздел “Записи” в главе 4) является специальным видом класса, который предназначен для эффективной работы с неизменяемыми данными. Его наиболее характерная особенность заключается в том, что он поддерживает *неразрушающее изменение* через новое ключевое слово (`with`):

```
Point p1 = new Point (2, 3);
Point p2 = p1 with { Y = 4 }; // p2 - копия p1, но с полем Y, установленным в 4
Console.WriteLine (p2);      // Point { X = 2, Y = 4 }

record Point
{
    public Point (double x, double y) => (X, Y) = (x, y);
    public double X { get; init; }
    public double Y { get; init; }
}
```

В простых случаях запись позволяет также избавиться от написания стереотипного кода определения свойств, конструктора и деструктора. Определение записи `Point` можно заменить следующим определением, не утрачивая функциональности:

```
record Point (double X, double Y);
```

Как и кортежи, по умолчанию записи поддерживают структурную эквивалентность. Записи могут быть подклассами других записей и включать конструкции, которые допускаются в классах. Во время выполнения компилятор реализует записи в виде классов.

Улучшения в сопоставлении с образцом

Реляционный шаблон (см. раздел “Шаблоны” в главе 4) разрешает применять в шаблонах операции `<`, `>`, `<=` и `>=`:

```
string GetWeightCategory (decimal bmi) => bmi switch {
    < 18.5m => "underweight",
    < 25m => "normal",
    < 30m => "overweight",
    _ => "obese" };
```

С помощью комбинаторов шаблонов шаблоны можно объединять посредством трех новых ключевых слов (`and`, `or` и `not`):

```
bool IsVowel (char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';
bool IsLetter (char c) => c is >= 'a' and <= 'z'
                           or >= 'A' and <= 'Z';
```

Подобно операциям `&&` и `||` комбинатор `and` имеет более высокий приоритет, чем комбинатор `or`. Приоритеты можно переопределять с использованием круглых скобок.

Комбинатор `not` можно использовать с *шаблоном типа* для проверки, имеет объект указанный тип или нет:

```
if (obj is not string) ...
```

Выражения `new` целевого типа

При конструировании объекта в C# 9 разрешено опускать имя типа, если компилятор способен однозначно его вывести:

```
System.Text.StringBuilder sb1 = new();
System.Text.StringBuilder sb2 = new ("Test");
```

Это особенно удобно, когда объявление и инициализация переменной находятся в разных частях кода:

```
class Foo
{
    System.Text.StringBuilder sb;
    public Foo (string initialValue) => sb = new (initialValue);
}
```

И в следующем сценарии:

```
MyMethod (new ("test"));
void MyMethod (System.Text.StringBuilder sb) { ... }
```

Дополнительные сведения ищите в разделе “Выражения `new` целевого типа” главы 2.

Улучшения в возможностях взаимодействия

В C# 9 появились *указатели на функции* (см. раздел “Указатели на функции” в главе 4 и раздел “Обратные вызовы с помощью указателей на функции” в главе 24). Основная их цель — позволить неуправляемому коду вызывать статические методы в C# без накладных расходов на создание экземпляра делегата, с возможностью обхода уровня P/Invoke, когда типы аргументов и возвращаемые типы являются *преобразуемыми* (*blittable*), т.е. представляются идентично с каждой стороны.

В C# 9 также введены целочисленные типы с собственным размером `nint` и `nuint` (см. раздел “Целочисленные типы с собственным размером” в главе 4), которые во время выполнения отображаются на `System.IntPtr` и `System.UIntPtr`. На этапе компиляции они ведут себя подобно числовым типам с поддержкой арифметических операций.

Остальные новые средства

В добавок версия C# 9 позволяет:

- переопределять метод или свойство, доступное только для чтения, чтобы возвращался более производный тип (см. раздел “Ковариантные возвращаемые типы” в главе 3);
- применять атрибуты к локальным функциям (см. раздел “Атрибуты” в главе 4);
- применять ключевое слово `static` к лямбда-выражениям или локальным функциям для гарантирования того, что не произойдет случайный захват локальных переменных или переменных экземпляра (см. раздел “Статические лямбда-выражения” в главе 4);
- заставлять любой тип работать с оператором `foreach` за счет написания расширяющего метода `GetEnumerator`;
- определять метод *инициализатора модуля*, который выполняется один раз при первой загрузке сборки, применяя атрибут `[ModuleInitializer]` к статическому методу `void` без параметров;
- использовать “отбрасывание” (символ подчеркивания) в качестве аргумента лямбда-выражения;
- создавать *расширенные частичные методы*, которые обязательны для реализации, делая возможными сценарии, такие как новые генераторы исходного кода Roslyn (см. раздел “Расширенные частичные методы” в главе 3);
- применять атрибут к методам, типам или модулям, чтобы предотвращать инициализацию локальных переменных исполняющей средой (см. раздел “Атрибут `SkipLocalsInit`” в главе 4).

Нововведения версии C# 8.0

Версия C# 8.0 впервые поставлялась вместе с *Visual Studio 2019* и по-прежнему используется в наши дни при нацеливании на .NET Core 3 или .NET Standard 2.1.

Индексы и диапазоны

Индексы и диапазоны упрощают работу с элементами или порциями массива (либо с низкоуровневыми типами `Span<T>` и `ReadOnlySpan<T>`).

Индексы позволяют ссылаться на элементы относительно конца массива с применением операции `^`. Например, `^1` ссылается на последний элемент, `^2` ссылается на предпоследний элемент и т.д.:

```
char[] vowels = new char[] {'a','e','i','o','u'};  
char lastElement = vowels [^1]; // 'u'  
char secondToLast = vowels [^2]; // 'o'
```

Диапазоны позволяют “нарезать” массив посредством операции . . .

```
char[] firstTwo = vowels [..2];           // 'a', 'e'  
char[] lastThree = vowels [2..];          // 'i', 'o', 'u'  
char[] middleOne = vowels [2..3]          // 'i'  
char[] lastTwo =    vowels [^2..];         // 'o', 'u'
```

Индексы и диапазоны реализованы в C# с помощью типов Index и Range:

```
Index last = ^1;  
Range firstTwoRange = 0..2;  
char[] firstTwo = vowels [firstTwoRange]; // 'a', 'e'
```

Вы можете поддерживать индексы и диапазоны в собственных классах, определяя индексатор с типом параметра Index или Range:

```
class Sentence  
{  
    string[] words = "The quick brown fox".Split();  
    public string this [Index index] => words [index];  
    public string[] this [Range range] => words [range];  
}
```

За дополнительной информацией обращайтесь в раздел “Индексы и диапазоны” главы 2.

Присваивание с объединением с null

Операция ??= присваивает значение переменной, только если она равна null. Вместо кода:

```
if (s == null) s = "Hello, world";
```

теперь можно записывать такой код:

```
s ??= "Hello, world";
```

Объявления using

Если опустить круглые скобки и блок операторов, следующий за оператором using, то он становится *объявлением using*. В результате ресурс освобождается, когда поток управления выходит за пределы включающего блока операторов:

```
if (File.Exists ("file.txt"))  
{  
    using var reader = File.OpenText ("file.txt");  
    Console.WriteLine (reader.ReadLine());  
    ...  
}
```

В этом случае память для reader будет освобождена, как только поток выполнения окажется вне блока оператора if.

Члены, предназначенные только для чтения

В C# 8 разрешено применять модификатор readonly к функциям структуры. Это гарантирует, что если функция попытается модифицировать любое поле, то генерируется ошибка на этапе компиляции:

```
struct Point
{
    public int X, Y;
    public readonly void ResetX() => X = 0; // Ошибка!
}
```

Если функция `readonly` вызывает функцию не `readonly`, тогда компилятор генерирует предупреждение (и защитным образом копирует структуру во избежание возможности изменения).

Статические локальные методы

Добавление модификатора `static` к локальному методу не позволяет ему видеть локальные переменные и параметры объемлющего метода, что помогает ослабить связность и разрешить локальному методу объявлять переменные по своему усмотрению без риска возникновения конфликта с переменными в объемлющем методе.

Стандартные члены интерфейса

В C# 8 к члену интерфейса можно добавлять стандартную реализацию, делая его необязательным для реализации:

```
interface ILogger
{
    void Log (string text) => Console.WriteLine (text);
}
```

В итоге появляется возможность добавлять член к интерфейсу, не нарушая работу реализаций. Стандартные реализации должны вызываться явно через интерфейс:

```
((ILogger)new Logger()).Log ("message");
```

В интерфейсах также можно определять статические члены (включая поля), к которым возможен доступ из кода внутри стандартных реализаций:

```
interface ILogger
{
    void Log (string text) => Console.WriteLine (Prefix + text);
    static string Prefix = "";
}
```

либо из кода за рамками интерфейса, если только доступ к ним не ограничен посредством модификатора доступности для статического члена интерфейса (такого как `private`, `protected` или `internal`):

```
ILogger.Prefix = "File log: ";
```

Поля экземпляров запрещены. Дополнительные сведения ищите в разделе “Стандартные члены интерфейса” главы 3.

Выражения `switch`

Начиная с версии C# 8, конструкцию `switch` можно использовать в контексте выражения:

```

string cardName = cardNumber switch // Предполагается, что cardNumber -
                                    // целое число
{
    13 => "King",                // Король
    12 => "Queen",               // Дама
    11 => "Jack",                // Валет
    _   => "Pip card"           // Нефигурная карта; эквивалент default
};

```

Дополнительные примеры приведены в разделе “Выражения switch” главы 2.

Шаблоны кортежей, позиционные шаблоны и шаблоны свойств

В C# 8 поддерживаются три новых шаблона по большей части в интересах операторов/выражений `switch` (см. раздел “Шаблоны” в главе 4). Шаблоны кортежей позволяют переключаться по множеству значений:

```

int cardNumber = 12; string suite = "spades";
string cardName = (cardNumber, suite) switch
{
    (13, "spades") => "King of spades",
    (13, "clubs")  => "King of clubs",
    ...
};

```

Позиционные шаблоны допускают похожий синтаксис для объектов, которые предоставляют деконструктор, а *шаблоны свойств* позволяют делать сопоставление со свойствами объекта. Все шаблоны можно применять как в операторах/выражениях `switch`, так и в операции `is`. В следующем примере используется шаблон свойств для проверки, является ли `obj` строкой с длиной 4:

```
if (obj is string { Length: 4 }) ...
```

Ссылочные типы, допускающие null

В то время как *типы значений*, *допускающие null*, наделяют типы значений способностью принимать значение `null`, *ссылочные типы, допускающие null*, делают противоположное и привносят в ссылочные типы возможность (в определенной степени) *не быть null*, что помогает предотвращать ошибки `NullReferenceException`. Ссылочные типы, допускающие `null`, обеспечивают уровень безопасности, который навязывается исключительно компилятором в форме предупреждений и ошибок, когда он обнаруживает код, подверженный риску генерации `NullReferenceException`.

Ссылочные типы, допускающие `null`, могут быть включены либо на уровне проекта (через элемент `Nullable` в файле проекта `.csproj`), либо в коде (через директиву `#nullable`). После их включения компилятор делает недопустимость `null` правилом по умолчанию: если вы хотите, чтобы ссылочный тип принимал значения `null`, тогда должны применять суффикс `?` для указания *ссылочного типа, допускающего null*:

```

(nullable enable      // Начиная с этого места, включить
     // ссылочные типы, допускающие null
string s1 = null;    // Компилятор генерирует предупреждение!
                     // (s1 не допускает null)
string? s2 = null;  // Нормально: s2 имеет ссылочный тип, допускающий null

```

Неинициализированные поля также приводят к генерации предупреждения (если тип не помечен как допускающий null), как и разыменование ссылочного типа, допускающего null, если компилятор считает, что может возникнуть исключение NullReferenceException:

```
void Foo (string? s) => Console.Write (s.Length); // Предупреждение (.Length)
```

Чтобы убрать предупреждение, можно использовать *null-терпимую* (*null-forgiving*) *операцию* (!):

```
void Foo (string? s) => Console.Write (s!.Length);
```

Полное обсуждение ищите в разделе “Ссылочные типы, допускающие null” главы 4.

Асинхронные потоки данных

До выхода версии C# 8 вы могли применять `yield return` для написания *итератора* или `await` для написания *асинхронной функции*. Но делать и то, и другое с целью написания итератора, который ожидает, асинхронно выдавая элементы, было невозможно. В версии C# 8 ситуация исправлена за счет введения *асинхронных потоков данных*:

```
async IAsyncEnumerable<int> RangeAsync (
    int start, int count, int delay)
{
    for (int i = start; i < start + count; i++)
    {
        await Task.Delay (delay);
        yield return i;
    }
}
```

Оператор `await foreach` потребляет асинхронный поток данных:

```
await foreach (var number in RangeAsync (0, 10, 100))
    Console.WriteLine (number);
```

За дополнительными сведениями обращайтесь в раздел “Асинхронные потоки данных” главы 14.

Нововведения версий C# 7.x

Версии C# 7.x впервые поставлялись вместе с Visual Studio 2017. В настоящее время версия C# 7.3 все еще используется в Visual Studio 2019 при нацеливании на .NET Core 2, .NET Framework 4.6–4.8 или .NET Standard 2.0.

C# 7.3

В C# 7.3 произведены незначительные улучшения существующих средств, такие как возможность использования операций эквивалентности с кортежами, усовершенствованное распознавание перегруженных версий и возможность применения атрибутов к поддерживающим полям автоматических свойств:

```
[field:NonSerialized]
public int MyProperty { get; set; }
```

Кроме того, версия C# 7.3 построена на основе расширенных программных средств низкоуровневого выделения памяти C# 7.2 с возможностью повторного присваивания значений локальным ссылочным переменным (`ref local`) без необходимости в закреплении при индексации полей `fixed` и поддержкой инициализаторов полей с помощью `stackalloc`:

```
int* pointer = stackalloc int[] {1, 2, 3};  
Span<int> arr = stackalloc [] {1, 2, 3};
```

Обратите внимание, что память, распределенная в стеке, может присваиваться прямо `Span<T>`. Мы опишем интервалы и причины их использования в главе 23.

C# 7.2

В C# 7.2 были добавлены модификатор `private protected` (*пересечение internal и protected*), возможность указания именованных аргументов с позиционными аргументами при вызове методов, а также структуры `readonly`. Структура `readonly` обязывает все поля быть `readonly`, чтобы помочь заявить о намерении и предоставить компилятору большую свободу в оптимизации:

```
readonly struct Point  
{  
    public readonly int X, Y; // X и Y обязаны быть readonly  
}
```

В C# 7.2 также добавлены специализированные возможности, содействующие микрооптимизации и программированию с низкоуровневым выделением памяти: см. разделы “Модификатор `in`”, “Локальные ссылочные переменные” и “Возвращаемые ссылочные значения” в главе 2 и раздел “Ссылочные структуры” в главе 3.

C# 7.1

Начиная с версии C# 7.1, в случае применения ключевого слова `default` тип допускается не указывать, если он может быть выведен:

```
decimal number = default; // number является десятичным
```

Кроме того, C# 7.1 ослабляет правила для операторов `switch` (так что для параметров обобщенных типов можно использовать сопоставление с образцом), позволяет методу `Main` программы быть асинхронным и разрешает выводить имена элементов кортежей:

```
var now = DateTime.Now;  
var tuple = (now.Hour, now.Minute, now.Second);
```

Усовершенствования числовых литералов

Для улучшения читабельности числовые литералы в C# 7 могут включать символы подчеркивания, называемые *разделителями групп разрядов*, которые компилятор игнорирует:

```
int million = 1_000_000;
```

С помощью префикса `0b` могут указываться *двоичные литералы*:

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

Переменные `out` и отбрасывание

В версии C# 7 облегчен вызов методов, содержащих параметры `out`. Прежде всего, теперь вы можете объявлять *переменные out* на лету (см. раздел “Переменные `out` и отбрасывание” в главе 2):

```
bool successful = int.TryParse ("123", out int result);
Console.WriteLine (result);
```

А при вызове метода с множеством параметров `out` с помощью символа подчеркивания вы можете *отбрасывать* те из них, которые вам не интересны:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
Console.WriteLine (x);
```

Шаблоны типов и шаблонные переменные

Посредством операции `is` вы также можете вводить переменные на лету. Они называются *шаблонными переменными* (см. раздел “Введение шаблонной переменной” в главе 3):

```
void Foo (object x)
{
    if (x is string s)
        Console.WriteLine (s.Length);
}
```

Оператор `switch` поддерживает шаблоны типов, поэтому вы можете переключаться на основе *типа* и на основе констант (см. раздел “Переключение по типам” в главе 2). Вы можете указывать условия в конструкции `when` и также переключаться по значению `null`:

```
switch (x)
{
    case int i:
        Console.WriteLine ("It's an int!"); // Это целочисленное значение!
        break;
    case string s:
        Console.WriteLine (s.Length); // Можно использовать переменную s
        break;
    case bool b when b == true: // Соответствует, только когда b равно true
        Console.WriteLine ("True");
        break;
    case null:
        Console.WriteLine ("Nothing"); // Ничего
        break;
}
```

Локальные методы

Локальный метод — это метод, объявленный внутри другой функции (см. раздел “Локальные методы” в главе 3):

```
void WriteCubes()
{
    Console.WriteLine (Cube (3));
    Console.WriteLine (Cube (4));
```

```
Console.WriteLine (Cube (5));
int Cube (int value) => value * value * value;
}
```

Локальные методы видны только вмещающей функции и могут захватывать локальные переменные тем же способом, что и лямбда-выражения.

Больше членов, сжатых до выражений

В версии C# 6 появился синтаксис сжатия до выражений (`=>`) для методов, свойств только для чтения, операций и индексаторов. В версии C# 7 он был расширен на конструкторы, свойства для чтения/записи и финализаторы:

```
public class Person
{
    string name;

    public Person (string name) => Name = name;
    public string Name
    {
        get => name;
        set => name = value ?? "";
    }

    ~Person () => Console.WriteLine ("finalize"); // финализировать
}
```

Деконструкторы

В версии C# 7 появился шаблон *деконструирования* (см. раздел “Деконструкторы” в главе 3). В то время как конструктор обычно принимает набор значений (в качестве параметров) и присваивает их полям, деконструктор делает противоположное, присваивая поля обратно набору переменных. Деконструктор для класса Person из предыдущего примера можно было бы написать следующим образом (обработка исключений опущена):

```
public void Deconstruct (out string firstName, out string lastName)
{
    int spacePos = name.IndexOf (' ');
    firstName = name.Substring (0, spacePos);
    lastName = name.Substring (spacePos + 1);
}
```

Деконструкторы вызываются с помощью специального синтаксиса:

```
var joe = new Person ("Joe Bloggs");
var (first, last) = joe; // Деконструирование
Console.WriteLine (first); // Joe
Console.WriteLine (last); // Bloggs
```

Кортежи

Пожалуй, самым заметным усовершенствованием, внесенным в версию C# 7, стала явная поддержка *кортежей* (см. раздел “Кортежи” в главе 4). Кортежи предоставляют простой способ хранения набора связанных значений:

```
var bob = ("Bob", 23);
Console.WriteLine (bob.Item1);                                // Bob
Console.WriteLine (bob.Item2);                                // 23
```

Кортежи в C# являются “синтаксическим сахаром” для использования обобщенных структур `System.ValueTuple<...>`. Но благодаря магии компилятора элементы кортежа могут быть именованными:

```
var tuple = (name:"Bob", age:23);
Console.WriteLine (tuple.name);                            // Bob
Console.WriteLine (tuple.age);                            // 23
```

С появлением кортежей у функций появилась возможность возвращения множества значений без обращения к параметрам `out` или добавочному типу:

```
static (int row, int column) GetFilePosition() => (3, 10);
static void Main()
{
    var pos = GetFilePosition();
    Console.WriteLine (pos.row);                           // 3
    Console.WriteLine (pos.column);                      // 10
}
```

Кортежи неявно поддерживают шаблон деконструирования, поэтому их легко деконструировать в индивидуальные переменные:

```
static void Main()
{
    (int row, int column) = GetFilePosition();           // Создает две локальные
                                                        // переменные
    Console.WriteLine (row);                            // 3
    Console.WriteLine (column);                        // 10
}
```

Выражения `throw`

До выхода версии C# 7 конструкция `throw` всегда была оператором. Теперь она может также появляться как выражение в функциях, сжатых до выражения:

```
public string Foo() => throw new NotImplementedException();
```

Выражение `throw` может также находиться внутри тернарной условной операции:

```
string Capitalize (string value) =>
    value == null ? throw new ArgumentException ("value") :
    value == "" ? "":
    char.ToUpper (value[0]) + value.Substring (1);
```

Нововведения версии C# 6.0

Особенностью версии C# 6.0, поставляемой в составе *Visual Studio 2015*, стал компилятор нового поколения, который был реализован полностью на языке C#.

Известный как проект Roslyn, новый компилятор делает видимым весь конвейер компиляции через библиотеки, позволяя проводить кодовый анализ для произвольного исходного кода. Сам компилятор представляет собой проект с открытым кодом, и его исходный код доступен по ссылке <http://github.com/dotnet/roslyn>.

Кроме того, в версии C# 6.0 появилось несколько небольших, но важных усовершенствований, направленных главным образом на сокращение беспорядка в коде.

null-условная операция (или элвис-операция), рассматриваемая в главе 2, устраняет необходимость в явной проверке на равенство null перед вызовом метода или доступом к члену типа. В следующем примере вместо генерации исключения NullReferenceException переменная result получает значение null:

```
System.Text.StringBuilder sb = null;
string result = sb?.ToString(); // Переменная result равна null
```

Функции, сжатые до выражений (expression-bodied function), которые обсуждаются в главе 3, дают возможность записывать методы, свойства, операции и индексаторы, содержащие единственное выражение, более компактно в стиле лямбда-выражений:

```
public int TimesTwo (int x) => x * 2;
public string SomeProperty => "Property value";
```

Инициализаторы свойств (глава 3) позволяют присваивать начальные значения автоматическим свойствам:

```
public DateTime Created { get; set; } = DateTime.Now;
```

Инициализируемые свойства также могут быть допускающими только чтение:

```
public DateTime Created { get; } = DateTime.Now;
```

Свойства, предназначенные только для чтения, можно также устанавливать в конструкторе, что облегчает создание неизменяемых (допускающих только чтение) типов.

Инициализаторы индексов (глава 4) делают возможной инициализацию за один шаг для любого типа, который открывает доступ к индексатору:

```
new Dictionary<int, string>()
{
    [3] = "three",
    [10] = "ten"
}
```

Интерполяция строк (см. раздел “Строковый тип” в главе 2) предлагает лаконичную альтернативу вызову метода string.Format:

```
string s = $"It is {DateTime.Now.DayOfWeek} today";
```

Фильтры исключений (см. раздел “Операторы try и исключения” в главе 4) позволяют применять условия к блокам catch:

```
string html;
try
{
    html = await new HttpClient().GetStringAsync ("http://asef");
}
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
    ...
}
```

Директива `using static` (см. раздел “Пространства имен” в главе 2) позволяет импортировать все статические члены типа, так что такими членами можно пользоваться без уточнения имени типа:

```
using static System.Console;
...
WriteLine ("Hello, world");           // WriteLine вместо Console.WriteLine
```

Операция `nameof` (глава 3) возвращает имя переменной, типа или другого символа в виде строки, что препятствует нарушению работы кода, когда какой-то символ переименовывается в Visual Studio:

```
int capacity = 123;
string x = nameof (capacity);        // x имеет значение "capacity"
string y = nameof (Uri.Host);         // y имеет значение "Host"
```

Наконец, в C# 6.0 разрешено применять `await` внутри блоков `catch` и `finally`.

Нововведения версии C# 5.0

Крупным нововведением версии C# 5.0 была поддержка асинхронных функций с помощью двух новых ключевых слов, `async` и `await`. Асинхронные функции делают возможными асинхронные продолжения, которые облегчают написание быстрореагирующих и безопасных к потокам обогащенных клиентских приложений. Они также упрощают написание эффективных приложений с высоким уровнем параллелизма и интенсивным вводом-выводом, которые не связывают потоковый ресурс при выполнении операций. Асинхронные функции подробно рассматриваются в главе 14.

Нововведения версии C# 4.0

В C# 4.0 появились четыре основных усовершенствования.

- *Динамическое связывание* (главы 4 и 19) откладывает связывание — процесс распознавания типов и членов — с этапа компиляции до времени выполнения и полезно в сценариях, которые иначе требовали бы сложного кода рефлексии. Динамическое связывание также удобно при взаимодействии с динамическими языками и компонентами COM.
- *Необязательные параметры* (глава 2) позволяют функциям указывать стандартные значения параметров, так что в вызывающем коде аргументы можно опускать. Именованные аргументы дают возможность вызывающему коду идентифицировать аргумент по имени, а не по позиции.

- Правила *вариантности типов* в C# 4.0 были ослаблены (главы 3 и 4), так что параметры типа в обобщенных интерфейсах и обобщенных делегатах могут помечаться как *ковариантные* или *контравариантные*, делая возможными более естественные преобразования типов.
- *Взаимодействие с COM* (глава 24) в C# 4.0 было улучшено в трех отношениях. Во-первых, аргументы могут передаваться по ссылке без ключевого свойства `ref` (что особенно удобно в сочетании с необязательными параметрами). Во-вторых, сборки, которые содержат типы взаимодействия с COM, можно *связывать*, а не *ссылаться* на них. Связанные типы взаимодействия поддерживают эквивалентность типов, устранив необходимость в наличии *основных сборок взаимодействия* (Primary Interop Assembly) и положив конец мучениям с ведением версий и развертыванием. В-третьих, функции, которые возвращают COM-типы `variant` из связанных типов взаимодействия, отображаются на тип `dynamic`, а не `object`, ликвидировав потребность в приведении.

Нововведения версии C# 3.0

Средства, добавленные в версии C# 3.0, по большей части были сосредоточены на возможностях языка *интегрированных запросов* (Language Integrated Query — LINQ). Язык LINQ позволяет записывать запросы прямо внутри программы C# и статически проверять их корректность, при этом допуская запросы как к локальным коллекциям (вроде списков или документов XML), так и к удаленным источникам данных (наподобие баз данных). Средства, которые были добавлены в версию C# 3.0 для поддержки LINQ, включают в себя неявно типизированные локальные переменные, анонимные типы, инициализаторы объектов, лямбда-выражения, расширяющие методы, выражения запросов и деревья выражений.

Неявно типизированные локальные переменные (ключевое слово `var`; см. главу 2) позволяют опускать тип переменной в операторе объявления, разрешая компилятору выводить его самостоятельно. Это уменьшает беспорядок, а также делает возможными *анонимные типы* (глава 4), которые представляют собой простые классы, создаваемые на лету и обычно применяемые в финальном выводе запросов LINQ. Массивы тоже могут быть неявно типизированными (см. главу 2).

Инициализаторы объектов (глава 3) упрощают конструирование объектов, позволяя устанавливать свойства прямо в вызове конструктора. Инициализаторы объектов работают как с именованными, так и с анонимными типами.

Лямбда-выражения (глава 4) — это миниатюрные функции, создаваемые компилятором на лету, которые особенно удобны в “текучем” синтаксисе запросов LINQ (глава 8).

Расширяющие методы (глава 4) расширяют существующий тип новыми методами (не изменяя определение типа) и делают статические методы похожими на методы экземпляра. Операции запросов LINQ реализованы как расширяющие методы.

Выражения запросов (глава 8) предоставляют высокоуровневый синтаксис для написания запросов LINQ, которые могут быть существенно проще при работе с множеством последовательностей или переменных диапазонов.

Деревья выражений (глава 8) — это миниатюрные кодовые модели документных объектов (Document Object Model — DOM), которые описывают лямбда-выражения, присвоенные переменным специального типа `Expression<TDelegate>`. Деревья выражений позволяют запросам LINQ выполнять удаленно (например, на сервере базы данных), поскольку их можно анализировать и транслировать во время выполнения (скажем, в оператор SQL).

В C# 3.0 также были добавлены автоматические свойства и частичные методы.

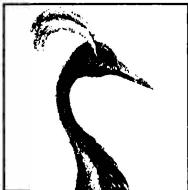
Автоматические свойства (глава 3) сокращают работу по написанию свойств, которые просто читают и устанавливают закрытое поддерживающее поле, заставляя компилятор делать все автоматически. *Частичные методы* (глава 3) позволяют автоматически сгенерированному частичному классу представлять настраиваемые привязки для ручного написания кода, который “исчезает” в случае, если не используется.

Нововведения версии C# 2.0

Крупными нововведениями в версии C# 2 были обобщения (глава 3), типы, допускающие значение `null` (глава 4), итераторы (глава 4) и анонимные методы (предшественники лямбда-выражений). Перечисленные средства подготовили почву для введения LINQ в версии C# 3.

В C# 2 также была добавлена поддержка частичных классов, статических классов и множества мелких разносторонних средств, таких как уточнитель псевдонима пространства имен, дружественные сборки и буферы фиксированных размеров.

Введение обобщений потребовало новой среды CLR (CLR 2.0), поскольку обобщения поддерживают полную точность типов во время выполнения.



Основы языка C#

В настоящей главе вы ознакомитесь с основами языка C#.



Почти все листинги кода, приведенные в настоящей книге, доступны в виде интерактивных примеров для LINQPad. Проработка примеров в сочетании с чтением книги ускоряет процесс изучения, т.к. вы можете редактировать код примеров и немедленно видеть результаты без необходимости в настройке проектов и решений в Visual Studio.

Для загрузки примеров перейдите на вкладку **Samples** (Примеры) в окне LINQPad и щелкните на ссылке **Download/import more samples** (Загрузить/импортировать дополнительные примеры). Утилита LINQPad бесплатна и доступна для загрузки по ссылке <http://www.linqpad.net>.

Первая программа на C#

Ниже показана программа, которая перемножает 12 и 30, после чего выводит на экран результат 360. Двойная косая черта (//) указывает на то, что остаток строки является *комментарием*:

```
int x = 12 * 30; // Оператор 1  
System.Console.WriteLine (x); // Оператор 2
```

Программа состоит из двух *операторов*. Операторы в C# выполняются последовательно и завершаются точкой с запятой. Первый оператор вычисляет выражение $12 * 30$ и сохраняет результат в *переменной* по имени x, которая имеет 32-битный целочисленный тип (int). Второй оператор вызывает *метод* `WriteLine` *класса* `Console`, который определен в пространстве имен `System`. В итоге значение переменной x выводится в текстовое окно на экране.

Метод выполняет функцию; класс группирует функции-члены и данные-члены с целью формирования объектно-ориентированного строительного блока. Класс `Console` группирует члены, которые поддерживают функциональность ввода-вывода в командной строке, такие как `WriteLine`. Класс является разновидностью типа, что будет обсуждаться в разделе “Основы типов” далее в главе.

На самом внешнем уровне типы организованы в *пространства имен*. Многие часто применяемые типы, включая класс `Console`, находятся в пространстве имен `System`. Библиотеки .NET организованы в виде вложенных пространств имен. Например, пространство имен `System.Text` содержит типы для обработки текста, а `System.IO` — типы для ввода-вывода.

Уточнение класса `Console` пространством имен `System` приводит к перегруженности кода. Директива `using` позволяет избежать такой перегруженности, импортируя пространство имен:

```
using System; // Импортировать пространство имен System.  
int x = 12 * 30;  
Console.WriteLine (x); // Нет необходимости указывать System.
```

Базовая форма многократного использования кода предусматривает написание функций более высокого уровня, которые вызывают функции более низкого уровня. Мы можем провести *рефакторинг* программы, выделив пригодный к многократному применению метод по имени `FeetToInches`, который умножает целое число на 12:

```
using System;  
Console.WriteLine (FeetToInches (30)); // 360  
Console.WriteLine (FeetToInches (100)); // 1200  
int FeetToInches (int feet)  
{  
    int inches = feet * 12;  
    return inches;  
}
```

Наш метод содержит последовательность операторов, заключенных в пару фигурных скобок. Такая конструкция называется *блоком операторов*. За счет указания *параметров* метод может получать *входные данные* из вызывающего кода, а за счет указания *возвращаемого типа* — передавать *выходные данные* обратно в вызывающий код. Наш метод `FeetToInches` имеет параметр для входного значения в футах и возвращаемый тип для выходного значения в дюймах:

```
int FeetToInches (int feet)  
...
```

Литералы 30 и 100 — это *аргументы*, передаваемые методу `FeetToInches`.

Если метод не принимает входные данные, то нужно использовать пустые круглые скобки. Если метод ничего не возвращает, тогда следует применять *ключевое слово void*:

```
using System;  
SayHello();  
void SayHello()  
{  
    Console.WriteLine ("Hello, world");  
}
```

Методы являются одним из нескольких видов функций в C#. Другим видом функции, задействованным в примере программы, была *операция **, которая выполняет умножение. Существуют также *конструкторы, свойства, события, индексаторы и финализаторы*.

Компиляция

Компилятор C# транслирует исходный код (набор файлов с расширением `.cs`) в *сборку* (*assembly*). Сборка — это единица упаковки и развертывания в .NET. Сборка может быть либо *приложением*, либо *библиотекой*. Нормальное

консольное или Windows-приложение имеет *точку входа*, тогда как библиотека — нет. Библиотека предназначена для вызова (ссылки) приложением или другими библиотеками. Сама платформа .NET представляет собой набор сборок (плюс исполняющую среду).

Программы в предыдущем разделе начинались непосредственно с последовательности операторов (называемых *операторами верхнего уровня*). Присутствие операторов верхнего уровня неявно создает точку входа для консольного или Windows-приложения. (Когда операторов верхнего уровня нет, точку входа приложения обозначает *метод Main* — см. раздел “Специальные типы” далее в главе.)



В отличие от сборок .NET Framework сборки .NET 8 не имеют расширения .exe. Файл .exe, который вы можете видеть после построения приложения .NET 8, является специфичным к платформе собственным загрузчиком, отвечающим за запуск сборки .dll вшего приложения. .NET 8 также позволяет создавать автономное развертывание, которое включает загрузчик, ваши сборки и обязательные части исполняющей среды .NET — все в одном файле .exe. Вдобавок .NET 8 допускает раннюю (AOT) компиляцию, при которой исполняемый файл содержит предварительно скомпилированный собственный код для более быстрого запуска и снижения потребления памяти.

Инструмент dotnet (dotnet.exe в Windows) помогает управлять исходным кодом .NET и двоичными сборками из командной строки. Вы можете применять его для построения и запуска своей программы в качестве альтернативы использованию интегрированной среды разработки (Integrated Development Environment — IDE), такой как Visual Studio или Visual Studio Code.

Вы можете получить инструмент dotnet, установив комплект .NET 8 SDK или Visual Studio. По умолчанию он находится в %ProgramFiles%\dotnet в Windows или в /usr/bin/dotnet в Ubuntu Linux.

Для компиляции приложения инструменту dotnet требуется *файл проекта*, а также один или большее количество файлов кода C#. Следующая команда генерирует шаблон нового консольного проекта (создает его базовую структуру):

```
dotnet new Console -n MyFirstProgram
```

Команда создает подкаталог по имени MyFirstProgram, содержащий файл проекта по имени MyFirstProgram.csproj и файл кода C# по имени Program.cs с методом, который выводит “Hello, world”.

Чтобы скомпилировать и запустить свою программу, введите приведенную ниже команду в подкаталоге MyFirstProgram:

```
dotnet run MyFirstProgram
```

Или введите такую команду, если вы хотите только скомпилировать программу, не запуская ее:

```
dotnet build MyFirstProgram.csproj
```

Выходная сборка будет сохранена в подкаталоге внутри bin\debug. Сборки подробно рассматриваются в главе 17.

Синтаксис

На синтаксис C# оказал влияние синтаксис языков С и С++. В этом разделе будут описаны элементы синтаксиса C# с применением в качестве примера следующей программы:

```
using System;  
int x = 12 * 30;  
Console.WriteLine (x);
```

Идентификаторы и ключевые слова

Идентификаторы представляют собой имена, которые программисты выбирают для своих классов, методов, переменных и т.д. Ниже перечислены идентификаторы из примера программы в порядке их появления:

```
System  x  Console  WriteLine
```

Идентификатор должен быть целостным словом, которое состоит из символов Unicode и начинается с буквы или символа подчеркивания. Идентификаторы в C# чувствительны к регистру символов. По соглашению для параметров, локальных переменных и закрытых полей должен применяться *“верблюжий” стиль* (вроде myVariable), а для всех остальных идентификаторов — *стиль Pascal* (наподобие MyMethod). Ключевые слова являются именами, которые имеют для компилятора особый смысл. В примере программы присутствуют два ключевых слова — using и int. Большинство ключевых слов *зарезервировано*, а это означает, что их нельзя использовать в качестве идентификаторов. Вот полный список зарезервированных ключевых слов в языке C#:

abstract	event	new	string
as	explicit	null	struct
base	extern	object	switch
bool	false	operator	this
break	finally	out	throw
byte	fixed	override	true
case	float	params	try
catch	for	private	typeof
char	foreach	protected	uint
checked	goto	public	ulong
class	if	readonly	unchecked
const	implicit	record	unsafe
continue	in	ref	ushort
decimal	int	return	using
default	interface	sbyte	virtual
delegate	internal	sealed	void
do	is	short	volatile
double	lock	sizeof	while
else	long	stackalloc	
enum	namespace	static	

Если вам действительно нужен идентификатор с именем, которое конфликтует с ключевым словом, то к нему необходимо добавить префикс @. Например:

```
int using = 123;           // Не допускается
int @using = 123;          // Разрешено
```

Символ @ не является частью самого идентификатора. Таким образом, @myVariable — то же самое, что и myVariable.

Контекстные ключевые слова

Некоторые ключевые слова являются **контекстными**, т.е. их можно использовать также в качестве идентификаторов — без символа @:

add	file	nameof	required
alias	from	nint	select
and	get	not	set
ascending	global	notnull	unmanaged
async	group	nuint	value
await	init	on	var
by	into	or	with
descending	join	orderby	when
dynamic	let	partial	where
equals	managed	remove	yield

Неоднозначность с контекстными ключевыми словами не может возникнуть внутри контекста, в котором они используются.

Литералы, знаки пунктуации и операции

Литералы — это элементарные порции данных, лексически встраиваемые в программу. В рассматриваемом примере программы присутствуют литералы 12 и 30.

Знаки пунктуации помогают размечать структуру программы. Примером может служить символ точки с запятой, который завершает оператор. Операторы могут записываться в нескольких строках:

```
Console.WriteLine
(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Операция преобразует и объединяет выражения. Большинство операций в C# обозначаются с помощью некоторого символа, скажем, * для операции умножения. Мы обсудим операции более подробно позже в главе. Ниже перечислены операции, задействованные в примере программы:

```
=    *    .    ()
```

Точка обозначает членство (или десятичную точку в числовых литералах). Круглые скобки используются при объявлении или вызове метода; пустые круглые скобки указываются, когда метод не принимает аргументов. (Позже в главе вы увидите, что круглые скобки имеют и другие предназначения.) Знак “равно” выполняет присваивание. (Двойной знак “равно”, ==, производит сравнение эквивалентности.)

Комментарии

В C# поддерживаются два разных стиля документирования исходного кода: *однострочные комментарии* и *многострочные комментарии*. Однострочный комментарий начинается с двойной косой черты и продолжается до конца строки, например:

```
int x = 3; // Комментарий относительно присваивания 3 переменной x
```

Многострочный комментарий начинается с символов `/*` и заканчивается символами `*/`, например:

```
int x = 3; /* Это комментарий, который
занимает две строчки */
```

В комментарии могут быть встроены XML-дескрипторы документации, которые объясняются в разделе “XML-документация” главы 4.

Основы типов

Тип определяет шаблон для значения. В следующем примере применяются два литерала типа `int` со значениями 12 и 30. Кроме того, объявляется *переменная* типа `int` по имени `x`:

```
int x = 12 * 30;
Console.WriteLine (x);
```



Поскольку в большинстве листингов кода, приведенных в этой книге, требуются типы из пространства имен `System`, впредь оператор `using System;` будет опускаться кроме случаев, когда иллюстрируется какая-то концепция, связанная с пространствами имен.

Переменная обозначает ячейку в памяти, которая с течением времени может содержать разные значения. Напротив, *константа* всегда представляет одно и то же значение (подробнее об этом — позже):

```
const int y = 360;
```

Все значения в C# являются *экземплярами* какого-то типа. Смысл значения и набор возможных значений, которые способна иметь переменная, определяются ее типом.

Примеры предопределенных типов

Предопределенные типы — это типы, которые имеют специальную поддержку в компиляторе. Тип `int` является предопределенным типом для представления набора целых чисел, которые умещаются в 32 бита памяти, от -2^{31} до $2^{31}-1$, и стандартным типом для числовых литералов в рамках указанного диапазона. С экземплярами типа `int` можно выполнять функции, например, арифметические:

```
int x = 12 * 30;
```

Еще один предопределенный тип в C# — `string`. Тип `string` представляет последовательность символов, такую как ".NET" или "http://oreilly.com". Со строками можно работать, вызывая для них функции следующим образом:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);           // HELLO WORLD
int x = 2024;
message = message + x.ToString();
Console.WriteLine (message);                // Hello world2024
```

В этом примере вызывается `x.ToString()` для получения строкового представления целочисленного значения `x`. Вызывать метод `ToString` можно для переменной практически любого типа. Предопределенный тип `bool` поддерживает в точности два возможных значения: `true` и `false`. Тип `bool` обычно используется для условного разветвления потока выполнения с помощью оператора `if`:

```
bool simpleVar = false;
if (simpleVar)
    Console.WriteLine ("This will not print");    // Это не выводится
int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
    Console.WriteLine ("This will print");        // Это будет выведено
```



В языке C# предопределенные типы (также называемые *встроеннымми типами*) распознаются по ключевым словам C#. Пространство имен `System` в .NET содержит много важных типов, которые не являются предопределенными в C# (например, `DateTime`).

Специальные типы

Точно так же, как можно записывать собственные методы, допускается создавать и свои типы. В следующем примере определяется специальный тип по имени `UnitConverter` — класс, который служит шаблоном для преобразования единиц:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
UnitConverter milesToFeetConverter = new UnitConverter (5280);
Console.WriteLine (feetToInchesConverter.Convert(30));    // 360
Console.WriteLine (feetToInchesConverter.Convert(100));   // 1200
Console.WriteLine (feetToInchesConverter.Convert(
    milesToFeetConverter.Convert(1)));                  // 63360
public class UnitConverter
{
    int ratio;                                // Поле
    public UnitConverter (int unitRatio)        // Конструктор
    {
        ratio = unitRatio;
    }
    public int Convert (int unit)               // Метод
    {
        return unit * ratio;
    }
}
```



В приведенном примере определение класса находится в том же файле, где располагаются операторы верхнего уровня. Поступать так законно — при условии, что сначала идут операторы верхнего уровня — и приемлемо при написании небольших тестовых программ. Стандартный подход для более крупных программ предусматривает помещение определения класса в отдельный файл, скажем, UnitConverter.cs.

Члены типа

Тип содержит *данные-члены* и *функции-члены*. Данными-членами в типе UnitConverter является поле по имени ratio. Функции-члены в типе UnitConverter — это метод Convert и конструктор UnitConverter.

Симметричность предопределенных и специальных типов

Привлекательный аспект языка C# связан с тем, что между предопределенными и специальными типами имеется лишь несколько отличий. Предопределенный тип int служит шаблоном для целых чисел. Он содержит данные — 32 бита — и предоставляет функции-члены, работающие с этими данными, такие как ToString. Аналогичным образом наш специальный тип UnitConverter действует в качестве шаблона для преобразований единиц. Он хранит данные — коэффициент (ratio) — и предоставляет функции-члены для работы с этими данными.

Конструкторы и создание экземпляров

Данные создаются путем *создания экземпляров* типа. Создавать экземпляры предопределенных типов можно просто за счет применения литерала вроде 12 или "Hello world". Экземпляры специального типа создаются через операцию new. Мы объявляли и создавали экземпляр типа UnitConverter с помощью следующего оператора:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```

Немедленно после того, как операция new создала объект, вызывается *конструктор* объекта для выполнения инициализации. Конструктор определяется подобно методу за исключением того, что вместо имени метода и возвращаемого типа указывается имя типа, к которому относится конструктор:

```
public UnitConverter (int unitRatio) { ratio = unitRatio; }
```

Члены экземпляра и статические члены

Данные-члены и функции-члены, которые оперируют на *экземпляре* типа, называются *членами экземпляра*. Примерами членов экземпляра могут служить метод Convert в типе UnitConverter и метод ToString в типе int. По умолчанию члены являются членами экземпляра.

Данные-члены и функции-члены, которые не оперируют на экземпляре типа, можно помечать как *статические* (static). Для ссылки на статический член за пределами его типа указывается имя *типа*, а не *экземпляра*. Примером может служить метод WriteLine класса Console. Из-за того, что он статический, мы используем вызов Console.WriteLine(), но не new Console().WriteLine().

(В действительности Console объявлен как *статический класс*, т.е. все его члены являются статическими; создавать экземпляры класса Console никогда не придется.)

В следующем коде поле экземпляра Name принадлежит экземпляру класса Panda, представляющего панд, тогда как поле Population относится к набору всех экземпляров класса Panda. Ниже создаются два экземпляра класса Panda, выводятся их клички (поле Name) и популяция (поле Population):

```
Panda p1 = new Panda ("Pan Dee");
Panda p2 = new Panda ("Pan Dah");

Console.WriteLine (p1.Name);           // Pan Dee
Console.WriteLine (p2.Name);           // Pan Dah
Console.WriteLine (Panda.Population);    // 2

public class Panda
{
    public string Name;                // Поле экземпляра
    public static int Population;       // Статическое поле
    public Panda (string n)            // Конструктор
    {
        Name = n;                     // Присвоить значение полю экземпляра
        Population = Population + 1; // Инкрементировать значение
                                         // статического поля Population
    }
}
```

Попытка вычисления p1.Population или Panda.Name приведет к возникновению ошибки на этапе компиляции.

Ключевое слово **public**

Ключевое слово **public** открывает доступ к членам со стороны других классов. Если бы в рассматриваемом примере поле Name класса Panda не было помечено как **public**, то оно стало бы закрытым, и доступ к нему извне класса оказался бы невозможным. Пометка члена как открытого (**public**) означает, что данный тип разрешает другим типам видеть этот член, а все остальное будет относиться к закрытым деталям реализации. В рамках объектно-ориентированной терминологии говорят, что открытые члены *инкапсулируют* закрытые члены класса.

Определение пространств имен

В крупных программах имеет смысл организовывать типы в виде пространств имен. Вот как определить класс Panda внутри пространства имен Animals:

```
using System;
using Animals;

Panda p = new Panda ("Pan Dee");
Console.WriteLine (p.Name);

namespace Animals
{
```

```
public class Panda
{
    ...
}
```

В этом примере также *импортируется* пространство имен Animals, чтобы операторы верхнего уровня могли получать доступ к его типам, не уточняя их. Без такого импортирования пришлось бы поступать следующим образом:

```
Animals.Panda p = new Animals.Panda ("Pan Dee");
```

Пространства имен подробно обсуждаются в конце главы (в разделе “Пространства имен”).

Определение метода Main

До сих пор во всем примерах применялись операторы верхнего уровня (нововведение версии C# 9).

Без операторов верхнего уровня простое консольное или Windows-приложение выглядело бы так:

```
using System;
class Program
{
    static void Main() // Точка входа в программу
    {
        int x = 12 * 30;
        Console.WriteLine (x);
    }
}
```

В отсутствие операторов верхнего уровня компилятор C# ищет статический метод по имени Main, который становится точкой входа. Метод Main можно определять внутри любого класса (и допускается существование только одного метода Main).

Метод Main может дополнительно возвращать целочисленное значение (вместо void) для исполняющей среды (причем ненулевое значение обычно указывает на ошибку). Кроме того, метод Main может необязательно принимать в качестве параметра массив строк (который будет заполняться аргументами, переданными исполняющему файлу). Например:

```
static int Main (string[] args) {...}
```



Массив (наподобие `string[]`) представляет фиксированное количество элементов определенного типа. Массивы указываются за счет помещения квадратных скобок после типа элементов. Они будут описаны в разделе “Массивы” далее в главе.

(Метод Main может быть объявлен асинхронным и возвращать объект Task или `Task<int>` для поддержки асинхронного программирования, как объясняется в главе 14.)

Операторы верхнего уровня

Операторы верхнего уровня (введенные в версии C# 9) позволяют избежать багажа статического метода Main и содержащего его класса. Файл с операторами верхнего уровня состоит из трех частей в следующем порядке.

1. (Необязательные) директивы using.
2. Последовательность операторов, необязательно смешанная с объявлением методов.
3. (Необязательные) объявления типов и пространства имен.

Вот пример:

```
using System;                                // Часть 1
Console.WriteLine ("Hello, world");           // Часть 2
void SomeMethod1() { ... }                   // Часть 2
Console.WriteLine ("Hello again!");           // Часть 2
void SomeMethod2() { ... }                   // Часть 2
class SomeClass { ... }                      // Часть 3
namespace SomeNamespace { ... }              // Часть 3
```

Поскольку CLR явно не поддерживает операторы верхнего уровня, компилятор транслирует такой код следующим образом:

```
using System;                                // Часть 1
static class Program$ // Специальное имя, генерированное компилятором
{
    static void Main$ (string[] args) // Специальное имя,
                                      // генерированное компилятором
    {
        Console.WriteLine ("Hello, world"); // Часть 2
        void SomeMethod1() { ... }       // Часть 2
        Console.WriteLine ("Hello again!"); // Часть 2
        void SomeMethod2() { ... }       // Часть 2
    }
}
class SomeClass { ... }                      // Часть 3
namespace SomeNamespace { ... }              // Часть 3
```

Обратите внимание, что весь код из части 2 помещен внутрь метода Main\$. Это значит, что SomeMethod1 и SomeMethod2 действуют как *локальные методы*. В разделе “Локальные методы” главы 3 мы обсудим все последствия, самое важное из которых заключается в том, что локальные методы (если только они не объявлены статическими) могут получать доступ к переменным, объявленным внутри содержащего метода:

```
int x = 3;
LocalMethod();
void LocalMethod() { Console.WriteLine (x); } // Можно получать доступ к x
```

Еще одно последствие состоит в том, что к методам верхнего уровня нельзя обращаться из других классов или типов.

Операторы верхнего уровня могут дополнительно возвращать целочисленное значение вызывающему модулю и получать доступ к “магической” переменной типа string [] по имени args, которая соответствует аргументам командной строки, переданным вызывающим компонентом.

Так как программа может иметь только одну точку входа, в проекте C# допускается наличие не более одного файла с операторами верхнего уровня.

Типы и преобразования

В C# возможны преобразования между экземплярами совместимых типов. Преобразование всегда создает новое значение из существующего. Преобразования могут быть либо *неявными*, либо *явными*: неявные преобразования происходят автоматически, в то время как явные преобразования требуют *приведения*. В следующем примере мы *неявно* преобразуем тип `int` в `long` (который имеет в два раза больше битов, чем `int`) и *явно* приводим тип `int` к `short` (который имеет в половину меньше битов, чем `int`):

```
int x = 12345;           // int - 32-битное целое
long y = x;              // Неявное преобразование в 64-битное целое
short z = (short)x;      // Явное преобразование в 16-битное целое
```

Неявные преобразования разрешены, когда удовлетворяются перечисленные ниже условия:

- компилятор может гарантировать, что они всегда будут проходить успешно;
- в результате преобразования никакая информация не утрачивается¹.

И наоборот, явные преобразования требуются, когда справедливо одно из следующих утверждений:

- компилятор не может гарантировать, что они всегда будут проходить успешно;
- в результате преобразования информация может быть утрачена.

(Если компилятор способен определить, что преобразование будет терпеть неудачу *всегда*, то оба вида преобразования запрещаются. Преобразования, в которых участвуют обобщения, в определенных обстоятельствах также могут потерпеть неудачу; об этом пойдет речь в разделе “Параметры типа и преобразования” главы 3.)



Числовые преобразования, которые мы только что видели, встроены в язык. Вдобавок в C# поддерживаются *ссылочные преобразования* и *упаковывающие преобразования* (см. главу 3), а также *специальные преобразования* (см. раздел “Перегрузка операций” в главе 4). Компилятор не навязывает упомянутые выше правила для специальных преобразований, поэтому неудачно спроектированные типы могут вести себя по-другому.

Типы значений и ссылочные типы

Все типы C# делятся на следующие категории:

- типы значений;
- ссылочные типы;
- параметры обобщенных типов;
- типы указателей.

¹ Небольшое предостережение: очень высокие значения `long` после преобразования в `double` теряют в точности.



В этом разделе будут описаны типы значений и ссылочные типы. Параметры обобщенных типов будут рассматриваться в разделе “Обобщения” главы 3, а типы указателей — в разделе “Небезопасный код и указатели” главы 4.

Типы значений включают большинство встроенных типов (а именно — все числовые типы, тип `char` и тип `bool`), а также специальные типы `struct` и `enum`.

Ссылочные типы включают все классы, массивы, делегаты и интерфейсы. (Сюда также входит предопределенный тип `string`.)

Фундаментальное отличие между типами значений и ссылочными типами связано с тем, каким образом они хранятся в памяти.

Типы значений

Содержимым переменной или константы, относящейся к типу значения, является просто значение. Например, содержимое встроенного типа значения `int` — 32 бита данных.

С помощью ключевого слова `struct` можно определить специальный тип значения (рис. 2.1):

```
public struct Point { public int X; public int Y; }
```

или более лаконично:

```
public struct Point { public int X, Y; }
```

Структура Point



Рис. 2.1. Экземпляр типа значения в памяти

Присваивание экземпляра типа значения всегда приводит к **копированию** этого экземпляра, например:

```
Point p1 = new Point();
p1.X = 7;

Point p2 = p1;           // Присваивание приводит к копированию
Console.WriteLine (p1.X); // 7
Console.WriteLine (p2.X); // 7

p1.X = 9;               // Изменить p1.X
Console.WriteLine (p1.X); // 9
Console.WriteLine (p2.X); // 7
```

На рис. 2.2 видно, что экземпляры `p1` и `p2` хранятся независимо друг от друга.

Структура Point

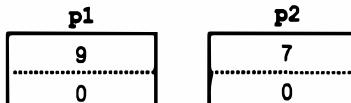


Рис. 2.2. Присваивание копирует экземпляр типа значения

Ссылочные типы

Ссылочный тип сложнее типа значения из-за наличия двух частей: *объекта* и *ссылки* на этот объект. Содержимым переменной или константы ссылочного типа является ссылка на объект, который содержит значение. Ниже приведен тип Point из предыдущего примера, переписанный в виде класса (рис. 2.3):

```
public class Point { public int X, Y; }
```



Рис. 2.3. Экземпляр ссылочного типа в памяти

Присваивание переменной ссылочного типа вызывает копирование ссылки, но не экземпляра объекта. В результате множество переменных могут ссылаться на один и тот же объект — то, что с типами значений обычно невозможно. Если повторить предыдущий пример при условии, что Point теперь является классом, тогда операция над p1 будет воздействовать на p2:

```
Point p1 = new Point();
p1.X = 7;

Point p2 = p1; // Копирует ссылку на p1
Console.WriteLine (p1.X); // 7
Console.WriteLine (p2.X); // 7

p1.X = 9; // Изменить p1.X
Console.WriteLine (p1.X); // 9
Console.WriteLine (p2.X); // 9
```

На рис. 2.4 видно, что p1 и p2 — две ссылки, указывающие на один и тот же объект.

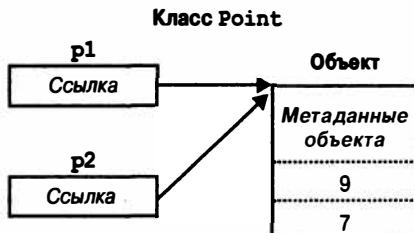


Рис. 2.4. Операция присваивания копирует ссылку

Значение null

Ссылке можно присваивать литерал `null`, который отражает тот факт, что ссылка не указывает на какой-либо объект:

```
Point p = null;
Console.WriteLine (p == null); // True

// Следующая строка вызывает ошибку времени выполнения
// (генерируется исключение NullReferenceException):
Console.WriteLine (p.X);

class Point {...}
```



В разделе “Сылочные типы, допускающие `null`” главы 4 будет описано средство языка C#, которое помогает сократить количество случайных ошибок `NullReferenceException`.

Напротив, тип значения обычно не может иметь значение `null`:

```
Point p = null; // Ошибка на этапе компиляции
int x = null; // Ошибка на этапе компиляции

struct Point {...}
```



В C# также имеется специальная конструкция под названием **типы значений, допускающие `null`**, которая предназначена для представления `null` в типах значений (см. раздел “Типы значений, допускающие `null`” в главе 4).

Накладные расходы, связанные с хранением

Экземпляры типов значений занимают в точности столько памяти, сколько необходимо для хранения их полей. В рассматриваемом примере `Point` требует 8 байтов памяти:

```
struct Point
{
    int x;           // 4 байта
    int y;           // 4 байта
}
```



Формально среда CLR располагает поля внутри типа по адресу, кратному размеру полей (выровненному максимум до 8 байтов). Таким образом, следующая структура в действительности потребляет 16 байтов памяти (с семью байтами после первого поля, которые “тратятся впустую”):

```
struct A { byte b; long l; }
```

Это поведение можно переопределить с помощью атрибута `StructLayout` (см. раздел “Отображение структуры на неуправляемую память” в главе 24).

Ссылочные типы требуют раздельного выделения памяти для ссылки и объекта. Объект потребляет столько памяти, сколько необходимо его полям, плюс дополнительный объем на административные нужды. Точный объем накладных расходов зависит от реализации исполняющей среды .NET, но составляет минимум 8 байтов, которые применяются для хранения ключа к типу объекта, а также временной информации, такой как его состояние блокировки при многопоточной обработке и флаг для указания, был ли объект закреплен, чтобы он не перемещался сборщиком мусора. Каждая ссылка на объект требует дополнительных 4 или 8 байтов в зависимости от того, на какой платформе функционирует исполняющая среда .NET — 32- или 64-разрядной.

Классификация предопределенных типов

Предопределенные типы в C# классифицируются следующим образом.

Типы значений

- Числовой
 - Целочисленный со знаком (`sbyte`, `short`, `int`, `long`)
 - Целочисленный без знака (`byte`, `ushort`, `uint`, `ulong`)
 - Вещественный (`float`, `double`, `decimal`)
- Булевский (`bool`)
- Символьный (`char`)

Ссылочные типы

- Строковый (`string`)
- Объектный (`object`)

Предопределенные типы C# являются псевдонимами типов .NET в пространстве имен `System`. Показанные ниже два оператора отличаются только синтаксисом:

```
int i = 5;  
System.Int32 i = 5;
```

Набор предопределенных типов значений, исключая `decimal`, известен в CLR как *примитивные типы*. Примитивные типы называются так оттого, что они поддерживаются непосредственно через инструкции в скомпилированном коде, которые обычно транслируются в прямую поддержку внутри имеющегося процессора, например:

```
// Лежащие в основе шестнадцатеричные представления  
int i = 7;           // 0x7  
bool b = true;       // 0x1  
char c = 'A';         // 0x41  
float f = 0.5f;       // Использует кодирование чисел с плавающей точкой IEEE
```

Типы `System.IntPtr` и `System.UIntPtr` также относятся к примитивным (см. главу 24).

Числовые типы

Предопределенные числовые типы C# показаны в табл. 2.1.

Таблица 2.1. Предопределенные числовые типы в C#

Тип C#	Тип в пространстве имен <code>System</code>	Суффикс	Размер в битах	Диапазон
Целочисленный со знаком				
sbyte	SByte		8	$-2^7 - 2^7 - 1$
short	Int16		16	$-2^{15} - 2^{15} - 1$
int	Int32		32	$-2^{31} - 2^{31} - 1$
long	Int64	L	64	$-2^{63} - 2^{63} - 1$
nint	IntPtr		32/64	
Целочисленный без знака				
byte	Byte		8	$0 - 2^8 - 1$
ushort	UInt16		16	$0 - 2^{16} - 1$
uint	UInt32	U	32	$0 - 2^{32} - 1$
ulong	UInt64	UL	64	$0 - 2^{64} - 1$
nuint	UIntPtr		32/64	
Вещественный				
float	Single	F	32	$\pm(\sim 10^{-45} - 10^{38})$
double	Double	D	64	$\pm(\sim 10^{-324} - 10^{308})$
decimal	Decimal	M	128	$\pm(\sim 10^{-28} - 10^{28})$

Из всех целочисленных типов `int` и `long` являются первоклассными типами, которым обеспечивается поддержка как в языке C#, так и в исполняющей среде. Другие целочисленные типы обычно применяются для реализации взаимодействия или когда главная задача связана с эффективностью хранения. Целочисленные типы с собственным размером `nint` и `nuint` наиболее полезны при работе с указателями, поэтому они будут описаны в разделе “Целочисленные типы с собственным размером” главы 4.

В рамках вещественных числовых типов `float` и `double` называются *типами с плавающей точкой*² и обычно используются в научных и графических вычислениях. Тип `decimal`, как правило, применяется в финансовых вычислениях, где требуется десятичная арифметика и высокая точность.

²Формально `decimal` — тоже тип с плавающей точкой, хотя в спецификации языка C# в таком качестве он не упоминается.



Платформа .NET дополняет этот список некоторыми специализированными числовыми типами, включая Int128 и UInt128 для 128-битных целых чисел со знаком и без знака, BigInteger для произвольно больших целых чисел и Half для 16-битных чисел с плавающей точкой. Тип Half предназначен главным образом для взаимодействия с процессорами графических плат и не имеет собственной поддержки в большинстве центральных процессоров, что делает типы float и double лучшими вариантами для общего применения.

Числовые литералы

Целочисленные литералы могут использовать десятичную или шестнадцатеричную форму записи; шестнадцатеричная форма записи предусматривает применение префикса 0x, например:

```
int x = 127;  
long y = 0x7F;
```

Вы можете вставлять символы подчеркивания в числовой литерал куда угодно, делая его более читабельным:

```
int million = 1_000_000;
```

Вы можете указывать числа в двоичном виде с помощью префикса 0b:

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

Вещественные литералы могут использовать десятичную и/или экспоненциальную форму записи:

```
double d = 1.5;  
double million = 1E06;
```

Выведение типа числового литерала

По умолчанию компилятор выводит тип числового литерала, относя его либо к double, либо к какому-то целочисленному типу.

Если литерал содержит десятичную точку или символ экспоненты (E), то он получает тип double.

В противном случае типом литерала будет первый тип, способный вместить значение литерала, из следующего списка: int, uint, long и ulong.

Например:

```
Console.WriteLine ( 1.0.GetType()); // Double (double)  
Console.WriteLine ( 1E06.GetType()); // Double (double)  
Console.WriteLine ( 1.GetType()); // Int32 (int)  
Console.WriteLine ( 0xF0000000.GetType()); // UInt32 (uint)  
Console.WriteLine (0x100000000.GetType()); // Int64 (long)
```

Числовые суффиксы

Числовые суффиксы явно определяют тип литерала. Суффиксы могут записываться либо строчными, либо прописными буквами; все они перечислены ниже.

Суффикс	Тип C#	Пример
F	float	float f = 1.0F;
D	double	double d = 1D;
M	decimal	decimal d = 1.0M;
U	uint	uint i = 1U;
L	long	long i = 1L;
UL	ulong	ulong i = 1UL;

Необходимость в суффиксах U и L возникает редко, поскольку типы uint, long и ulong почти всегда могут быть либо выведены, либо неявно преобразованы из int:

```
long i = 5; // Неявное преобразование без потерь литерала int в тип long
```

Суффикс D формально является избыточным из-за того, что все литералы с десятичной точкой выводятся в тип double. И к числовому литералу всегда можно добавить десятичную точку:

```
double x = 4.0;
```

Суффиксы F и M наиболее полезны и всегда должны применяться при указании литералов float или decimal. Без суффикса F следующая строка кода не скомпилируется, т.к. значение 4.5 выводится в тип double, для которого не существует неявного преобразования в тип float:

```
float f = 4.5F;
```

Тот же принцип справедлив для десятичных литералов:

```
decimal d = -1.23M; // Не скомпилируется без суффикса M
```

Семантика числовых преобразований подробно описана в следующем разделе.

Числовые преобразования

Преобразования между целочисленными типами

Преобразования между целочисленными типами являются *неявными*, когда целевой тип в состоянии представить каждое возможное значение исходного типа. В противном случае требуется явное преобразование, например:

```
int x = 12345;           // int - 32-битный целочисленный тип
long y = x;              // Неявное преобразование в 64-битный целочисленный тип
short z = (short)x;      // Явное преобразование в 16-битный целочисленный тип
```

Преобразования между типами с плавающей точкой

Тип float может быть неявно преобразован в double, т.к. double позволяет представить любое возможное значение float. Обратное преобразование должно быть явным.

Преобразования между типами с плавающей точкой и целочисленными типами

Все целочисленные типы могут быть неявно преобразованы во все типы с плавающей точкой:

```
int i = 1;  
float f = i;
```

Обратное преобразование обязано быть явным:

```
int i2 = (int)f;
```



Когда число с плавающей точкой приводится к целому, любая дробная часть отбрасывается; никакого округления не производится. Статический класс `System.Convert` предоставляет методы, которые выполняют преобразования между разнообразными числовыми типами с округлением (см. главу 6).

Неявное преобразование большого целочисленного типа в тип с плавающей точкой сохраняет *величину*, но иногда может приводить к потере *точности*. Причина в том, что типы с плавающей точкой всегда имеют большую величину, чем целочисленные типы, но могут иметь меньшую точность. Для демонстрации сказанного рассмотрим пример с более крупным числом:

```
int i1 = 100000001;  
float f = i1;           // Величина сохраняется, точность теряется  
int i2 = (int)f;       // 100000000
```

Десятичные преобразования

Все целочисленные типы могут быть неявно преобразованы в `decimal`, поскольку тип `decimal` в C# способен представлять любое возможное целочисленное значение. Все остальные числовые преобразования в тип `decimal` и из него должны быть явными, потому что они создают возможность выхода значения за пределы допустимого диапазона или потери точности.

Арифметические операции

Арифметические операции (+, -, *, /, %) определены для всех числовых типов кроме 8- и 16-битных целочисленных типов:

- + Сложение
- Вычитание
- * Умножение
- / Деление
- % Остаток от деления

Операции инкремента и декремента

Операции инкремента и декремента (++ и --) увеличивают и уменьшают значения переменных числовых типов на 1. Эти операции могут находиться до или после имени переменной в зависимости от того, когда требуется обновить значение переменной — до или после вычисления выражения; например:

```
int x = 0, y = 0;  
Console.WriteLine (x++); // Выводит 0; x теперь содержит 1  
Console.WriteLine (++y); // Выводит 1; y теперь содержит 1
```

Специальные операции с целочисленными типами

(К целочисленным типам относятся `int`, `uint`, `long`, `ulong`, `short`, `ushort`, `byte` и `sbyte`.)

Деление

Операции деления с целочисленными типами всегда отбрасывают остаток (округляют в направлении нуля). Деление на переменную, значение которой равно нулю, вызывает ошибку во время выполнения (исключение `DivideByZeroException`):

```
int a = 2 / 3; // 0  
int b = 0;  
int c = 5 / b; // Генерируется исключение DivideByZeroException
```

Деление на литерал или константу 0 генерирует ошибку на этапе компиляции.

Переполнение

Во время выполнения арифметические операции с целочисленными типами могут приводить к переполнению. По умолчанию это происходит молча — никакие исключения не генерируются, а результат демонстрирует поведение с циклическим возвратом, как если бы вычисление производилось над большим целочисленным типом с отбрасыванием дополнительных значащих битов. Например, декрементирование минимально возможного значения типа `int` дает в результате максимально возможное значение `int`:

```
int a = int.MinValue;  
a--;  
Console.WriteLine (a == int.MaxValue); // True
```

Операции проверки переполнения

Операция `checked` сообщает исполняющей среде о том, что вместо молчаливого переполнения она должна генерировать исключение `OverflowException`, когда выражение или оператор с целочисленным типом приводит к выходу за арифметические пределы этого типа. Операция `checked` воздействует на выражения с операциями `++, --, +, -` (бинарной и унарной), `*`, `/` и явными преобразованиями между целочисленными типами. Проверка переполнения сопряжена с небольшим снижением производительности.



Операция `checked` не оказывает никакого влияния на типы `double` и `float` (которые получают при переполнении специальные значения “бесконечности”, как вскоре будет показано) и на тип `decimal` (который проверяется всегда).

Операцию checked можно использовать либо с выражением, либо с блоком операторов:

```
int a = 1000000;
int b = 1000000;
int c = checked (a * b);           // Проверяет только это выражение
checked                         // Проверяет все выражения
{
    // в блоке операторов
    ...
    c = a * b;
    ...
}
```

Проверку на арифметическое переполнение можно сделать обязательной для всех выражений в программе, выбрав настройку checked на уровне проекта (в Visual Studio это делается на вкладке Advanced Build Settings (Дополнительные параметры сборки)). Если позже понадобится отключить проверку переполнения для конкретных выражений или операторов, тогда можно воспользоваться операцией unchecked. Например, следующий код не будет генерировать исключения, даже если выбрана настройка checked:

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

Проверка переполнения для константных выражений

Независимо от настройки checked проекта для выражений, вычисляемых во время компиляции, проверка переполнения производится всегда, если только не применена операция unchecked:

```
int x = int.MaxValue + 1;          // Ошибка на этапе компиляции
int y = unchecked (int.MaxValue + 1); // Ошибки отсутствуют
```

Побитовые операции

В C# поддерживаются следующие побитовые операции.

Операция	Описание	Пример выражения	Результат
~	Дополнение	~0xfU	0xffffffff0U
&	И	0xf0 & 0x33	0x30
	ИЛИ	0xf0 0x33	0xf3
^	Исключающее ИЛИ	0xff00 ^ 0xffff	0xf0f0
<<	Сдвиг влево	0x20 << 2	0x80
>>	Сдвиг вправо	0x20 >> 1	0x10
>>>	Беззнаковый сдвиг вправо	int.MinValue >>> 1	0x40000000

Операция сдвига вправо (>>) копирует старший бит при работе с целыми числами со знаком, тогда как операция беззнакового сдвига вправо (>>>) этого не делает.



Дополнительные побитовые операции предоставляются через класс `BitOperations` из пространства имен `System.Numerics` (см. раздел “Класс `BitOperations`” в главе 6).

8- и 16-битные целочисленные типы

К 8- и 16-битным целочисленным типам относятся `byte`, `sbyte`, `short` и `ushort`. В указанных типах отсутствуют собственные арифметические операции, а потому компилятор C# при необходимости неявно преобразует их в более крупные типы. Попытка присваивания результата переменной меньшего целочисленного типа может привести к получению ошибки на этапе компиляции:

```
short x = 1, y = 1;  
short z = x + y; // Ошибка на этапе компиляции
```

В данном случае переменные `x` и `y` неявно преобразуются в тип `int`, поэтому сложение может быть выполнено. Это означает, что результат тоже будет иметь тип `int`, который не может быть неявно приведен к типу `short` (из-за возможной потери информации). Чтобы такой код скомпилировался, потребуется добавить явное приведение:

```
short z = (short) (x + y); // Компилируется
```

Специальные значения `float` и `double`

В отличие от целочисленных типов типы с плавающей точкой имеют значения, которые определенные операции трактуют особым образом. Такими специальными значениями являются `NaN` (Not a Number — не число), $+\infty$, $-\infty$ и -0 . В классах `float` и `double` предусмотрены константы для `NaN`, $+\infty$ и $-\infty$, а также для других значений (`.MaxValue`, `.MinValue` и `Epsilon`), например:

```
Console.WriteLine (double.NegativeInfinity); // Минус бесконечность
```

Ниже перечислены константы, которые представляют специальные значения для типов `double` и `float`.

Специальное значение	Константа <code>double</code>	Константа <code>float</code>
<code>NaN</code>	<code>double.NaN</code>	<code>float.NaN</code>
$+\infty$	<code>double.PositiveInfinity</code>	<code>float.PositiveInfinity</code>
$-\infty$	<code>double.NegativeInfinity</code>	<code>float.NegativeInfinity</code>
-0	<code>-0.0</code>	<code>-0.0f</code>

Деление ненулевого числа на ноль дает в результате бесконечную величину:

```
Console.WriteLine ( 1.0 / 0.0); // Бесконечность  
Console.WriteLine (-1.0 / 0.0); // Минус бесконечность  
Console.WriteLine ( 1.0 / -0.0); // Минус бесконечность  
Console.WriteLine (-1.0 / -0.0); // Бесконечность
```

Деление нуля на ноль или вычитание бесконечности из бесконечности дает в результате NaN:

```
Console.WriteLine ( 0.0 / 0.0); // NaN  
Console.WriteLine ((1.0 / 0.0) - (1.0 / 0.0)); // NaN
```

Когда применяется операция ==, значение NaN никогда не будет равно другому значению, даже еще одному NaN:

```
Console.WriteLine (0.0 / 0.0 == double.NaN); // False
```

Для проверки, является ли значение специальным значением NaN, должен использоваться метод float.IsNaN или double.IsNaN:

```
Console.WriteLine (double.IsNaN (0.0 / 0.0)); // True
```

Однако в случае применения метода object.Equals два значения NaN равны:

```
Console.WriteLine (object.Equals (0.0 / 0.0, double.NaN)); // True
```



Значения NaN иногда удобны для представления специальных величин. Например, в Windows Presentation Foundation (WPF) с помощью double.NaN представлено измерение, значением которого является "Automatic" (автоматическое). Другой способ представления такого значения предусматривает использование типа, допускающего значение null (см. главу 4), а еще один способ — применение специальной структуры, которая служит оболочкой для числового типа с дополнительным полем (см. главу 3).

Типы float и double следуют спецификации IEEE 754 для формата представления чисел с плавающей точкой, которая поддерживается практически всеми процессорами. Подробную информацию относительно поведения этих типов можно найти на веб-сайте <http://www.ieee.org>.

Выбор между double и decimal

Тип double удобен в научных вычислениях (таких как расчет пространственных координат), а тип decimal — в финансовых вычислениях и для представления значений, которые являются *искусственными*, а не полученными в результате реальных измерений. Ниже представлен обзор отличий между типами double и decimal.

Характеристика	double	decimal
Внутреннее представление	Двоичное	Десятичное
Десятичная точность	15–16 значащих цифр	28–29 значащих цифр
Диапазон	$\pm(\sim 10^{-324} \dots \sim 10^{308})$	$\pm(\sim 10^{-28} \dots \sim 10^{28})$
Специальные значения	+0, -0, +∞, -∞ и NaN	Отсутствуют
Скорость обработки	Присущая процессору	Не присущая процессору (примерно в 10 раз медленнее, чем в случае double)

Ошибки округления вещественных чисел

Типы `float` и `double` внутренне представляют числа в двоичной форме. По указанной причине точно представляются только числа, которые могут быть выражены в двоичной системе счисления. На практике это означает, что большинство литералов с дробной частью (которые являются десятичными) не будут представлены точно, например:

```
float x = 0.1f; // Не точно 0.1
Console.WriteLine (x + x + x + x + x + x + x + x + x); // 1.0000001
```

Именно потому типы `float` и `double` не подходят для финансовых вычислений. В противоположность им тип `decimal` работает в десятичной системе счисления, так что он способен точно представлять дробные числа вроде `0.1`, выражимые в десятичной системе (а также в системах счисления с основаниями-множителями `10` — двоичной и пятеричной). Поскольку вещественные литералы являются десятичными, тип `decimal` может точно представлять такие числа, как `0.1`. Тем не менее, ни `double`, ни `decimal` не могут точно представлять дробное число с периодическим десятичным представлением:

```
decimal m = 1M / 6M; // 0.1666666666666666666666666667M
double d = 1.0 / 6.0; // 0.1666666666666666
```

Это приводит к накапливающимся ошибкам округления:

```
decimal notQuiteWholeM = m+m+m+m+m; // 1.00000000000000000000000000002M
double notQuiteWholeD = d+d+d+d+d; // 0.9999999999999998
```

которые нарушают работу операций эквивалентности и сравнения:

```
Console.WriteLine (notQuiteWholeM == 1M); // False
Console.WriteLine (notQuiteWholeD < 1.0); // True
```

Булевский тип и операции

Тип `bool` в C# (псевдоним типа `System.Boolean`) представляет логическое значение, которому может быть присвоен литерал `true` или `false`.

Хотя для хранения булевского значения достаточно только одного бита, исполняющая среда будет использовать один байт памяти, т.к. это минимальная порция, с которой исполняющая среда и процессор могут эффективно работать. Во избежание непродуктивных расходов пространства в случае массивов инфраструктура .NET предлагает в пространстве имен `System.Collections` класс `BitArray`, который позволяет задействовать по одному биту для каждого булевского значения в массиве.

Булевые преобразования

Приведения и преобразования из типа `bool` в числовые типы и наоборот не разрешены.

Операции сравнения и проверки равенства

Операции `==` и `!=` проверяют на предмет эквивалентности и неэквивалентности значения любого типа и всегда возвращают значение `bool`³. Типы значений обычно поддерживают очень простое понятие эквивалентности:

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine (x == y);           // False
Console.WriteLine (x == z);           // True
```

Для ссылочных типов эквивалентность по умолчанию основана на ссылке, а не на действительном значении лежащего в основе объекта (более подробно об этом речь пойдет в главе 6):

```
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);          // False
Dude d3 = d1;
Console.WriteLine (d1 == d3);          // True

public class Dude
{
    public string Name;
    public Dude (string n) { Name = n; }
}
```

Операции эквивалентности и сравнения, `==`, `!=`, `<`, `>`, `>=` и `<=`, работают со всеми числовыми типами, но должны осмотрительно применяться с вещественными числами (как было указано выше в разделе “Ошибки округления вещественных чисел”). Операции сравнения также работают с членами типа `enum`, сравнивая лежащие в их основе целочисленные значения. Это будет описано в разделе “Перечисления” главы 3.

Операции эквивалентности и сравнения более подробно объясняются в разделе “Перегрузка операций” главы 4, а также в разделах “Сравнение эквивалентности” и “Сравнение порядка” главы 6.

Условные операции

Операции `&&` и `||` реализуют условия *И* и *ИЛИ*. Они часто применяются в сочетании с операцией `!`, которая выражает условие *НЕ*. В показанном ниже примере метод `UseUmbrella` (брать ли зонт) возвращает `true`, если дождливо (`rainy`) или солнечно (`sunny`) при условии, что также не ветрено (`windy`):

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
    return !windy && (rainy || sunny);
```

Когда возможно, операции `&&` и `||` сокращают вычисления. Возвращаясь к предыдущему примеру, если ветрено (`windy`), тогда выражение `(rainy || sunny)`

³ Эти операции разрешено перегружать (см. главу 4), чтобы они возвращали тип, отличающийся от `bool`, но на практике так почти никогда не поступают.

даже не вычисляется. Сокращение вычислений играет важную роль в обеспечении выполнения выражений, таких как показанное ниже, без генерации исключения `NullReferenceException`:

```
if (sb != null && sb.Length > 0) ...
```

Операции `&` и `|` также реализуют условия *И* и *ИЛИ*:

```
return !windy & (rainy | sunny);
```

Их отличие состоит в том, что они *не сокращают вычисления*. По этой причине `&` и `|` редко используются в качестве операций сравнения.



В отличие от языков C и C++ операции `&` и `|` производят *булевские* сравнения (без сокращения вычислений), когда применяются к выражениям `bool`. Операции `&` и `|` выполняются как побитовые только в случае применения к числам.

Условная (тернарная) операция

Условная операция (чаще называемая *тернарной операцией*, т.к. она единственная принимает три операнда) имеет вид `q ? a : b`, где результатом является `a`, если условие `q` равно `true`, и `b` — в противном случае, например:

```
static int Max (int a, int b)
{
    return (a > b) ? a : b;
}
```

Условная операция особенно удобна в выражениях LINQ, рассматриваемых в главе 8.

Строки и символы

Тип `char` в C# (псевдоним типа `System.Char`) представляет символ Unicode и занимает 2 байта (UTF-16). Литерал `char` указывается в одинарных кавычках:

```
char c = 'A'; // Простой символ
```

Управляющие последовательности выражают символы, которые не могут быть представлены или интерпретированы буквально. Управляющая последовательность состоит из символа обратной косой черты, за которым следует символ со специальным смыслом; например:

```
char newLine = '\n';
char backSlash = '\\';
```

Символы управляющих последовательностей показаны в табл. 2.2.

Управляющая последовательность `\u` (или `\x`) позволяет указывать любой символ Unicode в виде его шестнадцатеричного кода, состоящего из четырех цифр:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Таблица 2.2. Символы управляющих последовательностей

Символ	Смысл	Значение
\'	Одинарная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\"	Обратная косая черта	0x005C
\0	Пусто	0x0000
\a	Сигнал внимания	0x0007
\b	Забой	0x0008
\f	Перевод страницы	0x000C
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальная табуляция	0x0009
\v	Вертикальная табуляция	0x000B

Символьные преобразования

Неявное преобразование `char` в числовой тип работает для числовых типов, которые могут вместить значение `short` без знака. Для других числовых типов требуется явное преобразование.

Строковый тип

Тип `string` в C# (псевдоним типа `System.String`, подробно рассматриваемый в главе 6) представляет неизменяемую последовательность символов Unicode. Строковый литерал указывается в двойных кавычках:

```
string a = "Heat";
```



`string` — это ссылочный тип, а не тип значения. Тем не менее, его операции эквивалентности следуют семантике типов значений:

```
string a = "test";
string b = "test";
Console.WriteLine(a == b); // True
```

Управляющие последовательности, допустимые для литералов `char`, также работают внутри строк:

```
string a = "Here's a tab:\t";
```

Платой за это является необходимость дублирования символа обратной косой черты, когда он нужен буквально:

```
string a1 = "\\\\"server\\fileshare\\helloworld.cs";
```

Чтобы избежать такой проблемы, в C# разрешены дословные строковые литералы. Дословный строковый литерал снабжается префиксом `@` и не поддержи-

вает управляющие последовательности. Следующая дословная строка идентична предыдущей строке:

```
string a2 = @"\\server\\fileshare\\helloworld.cs";
```

Дословный строковый литерал может также занимать несколько строк:

```
string escaped = "First Line\r\nSecond Line"; необрабатываем
string verbatim = @"First Line
Second Line";
```

```
//Выводит True, если в текстовом редакторе используются разделители строк CR-LF:
Console.WriteLine (escaped == verbatim);
```

Для включения в дословный строковый литерал символа двойной кавычки его понадобится записать дважды:

```
string xml = @"<customer id=""123""></customer>";
```

Необрабатываемые строковые литералы (C# 11)

В результате помещения строки в три и более символов кавычек (""""") создается *необрабатываемый строковый литерал*. Необрабатываемые строковые литералы могут содержать практически любую последовательность символов без служебных символов или удвоения:

```
string raw = """<file path="c:\\temp\\test.txt"></file>""";
```

Необрабатываемые строковые литералы упрощают представление литералов JSON, XML и HTML, а также регулярных выражений и исходного кода. Если необходимо поместить три или большее количество кавычек в саму строку, то можно заключить ее в четыре или более кавычек:

```
string raw = """"The """ sequence denotes raw string literals."""";
```

На многострочные необрабатываемые строковые литералы распространяются особые правила. Строку "Line 1\r\nLine 2" можно представить следующим образом:

```
string multiLineRaw = """
Line 1
Line 2
""";
```

Обратите внимание, что открывающие и закрывающие кавычки должны находиться в отдельных строках содержимого строки. Кроме того:

- пробельные символы после *открывающей последовательности """* (в той же самой строчке) игнорируются;
- пробельные символы, предшествующие *закрывающей последовательности """* (в той же самой строчке), трактуются как *общий отступ* и удаляются из каждой строчки в строке; это позволяет включать отступ для удобства чтения исходного кода, причем отступ не становится частью строки.

Вот еще один пример, иллюстрирующий правила использования многострочных необрабатываемых строковых литералов:

```
if(true) Console.WriteLine ("""
{
    "Name" : "Joe"
}
""");
```

Ниже показан вывод:

```
{
    "Name" : "Joe"
}
```

Компилятор сообщит об ошибке, если каждая строчка многострочного необрабатываемого строкового литерала не предварена общим отступом, указанным в закрывающих кавычках.

Необрабатываемые строковые литералы можно интерполировать при соблюдении специальных правил, описанных в разделе “Интерполяция строк” далее в главе.

Конкатенация строк

Операция + выполняет конкатенацию двух строк:

```
string s = "a" + "b";
```

Один из операндов может быть нестроковым значением; в этом случае для него будет вызван метод ToString:

```
string s = "a" + 5; // a5
```

Многократное применение операции + для построения строки является неэффективным: более удачное решение предусматривает использование типа System.Text.StringBuilder (описанного в главе 6).

Интерполяция строк

Строка, предваренная символом \$, называется *интерполированной строкой*. Интерполированные строки могут содержать выражения, заключенные в фигурные скобки:

```
int x = 4;
Console.Write($"A square has {x} sides"); // Выводит: A square has 4 sides
```

Внутри скобок может быть указано любое допустимое выражение C# произвольного типа, и компилятор C# преобразует это выражение в строку, вызывая ToString или эквивалентный метод данного типа. Форматирование можно изменять путем добавления к выражению двоеточия и *форматной строки* (форматные строки описаны в разделе “Метод string.Format и смешанные форматные строки” главы 6):

```
string s = $"255 in hex is {byte.MaxValue:X2}";
           // X2 - шестнадцатеричное значение с двумя цифрами
           // s получает значение "255 in hex is FF"
```

Если вам необходимо применять двоеточие для другой цели (скажем, в тернарной условной операции, которая будет раскрыта позже), тогда все выражение потребуется поместить в круглые скобки:

```
bool b = true;
Console.WriteLine ($"The answer in binary is {(b ? 1 : 0)}");
```

Начиная с версии C# 10, интерполированные строки могут быть константами при условии, что интерполированные значения являются константами:

```
const string greeting = "Hello";
const string message = $"{greeting}, world";
```

Начиная с версии C# 11, интерполированные строки можно разносить по нескольким строчкам (будь они стандартные или дословные):

```
string s = $"this interpolation spans {1 +
1} lines";
```

Необрабатываемые строковые литералы (начиная с версии C# 11) также могут быть интерполированными:

```
string s = $"""\nThe date and time is {DateTime.Now}""";
```

Чтобы включить фигурную скобку в интерполированную строку:

- при использовании стандартных и дословных строковых литералов повторите желаемый символ фигурной скобки;
- при использовании необрабатываемых строковых литералов измените последовательность интерполяции, повторив префикс \$.

```
Console.WriteLine ($$"""\nTimeStamp": "{DateTime.Now}" """);
// Вывод: { "TimeStamp": "01/01/2024 12:13:25 PM" }
```

Это сохраняет возможность копирования и вставки текста в необрабатываемый строковый литерал без необходимости изменения строки.

Сравнение строк

Для сравнения эквивалентности строк можно использовать операцию == (или один из методов Equals типа string). Для сравнения порядка необходимо применять метод CompareTo строки; операции < и > не поддерживаются. Сравнение эквивалентности и порядка рассматривается в разделе “Сравнение строк” главы 6.

Строки UTF-8

Начиная с версии C# 11, можно использовать суффикс u8 для создания строковых литералов, закодированных в UTF-8, а не в UTF-16. Данное средство предназначено для сложных сценариев, таких как низкоуровневая обработка текста JSON для повышения производительности:

```
ReadOnlySpan<byte> utf8 = "ab→cd"u8; // Символ стрелки занимает 3 байта
Console.WriteLine (utf8.Length); // 7
```

Лежащим в основе типом является `ReadOnlySpan<byte>`, который будет рассматриваться в главе 23. Переменную `utf8` можно преобразовать в массив, вызвав метод `ToArray()`.

Массивы

Массив представляет фиксированное количество переменных (называемых **элементами**) определенного типа. Элементы массива всегда хранятся в непрерывном блоке памяти, обеспечивая высокоэффективный доступ.

Массив обозначается квадратными скобками после типа элементов:

```
char[] vowels = new char[5]; // Объявить массив из 5 символов
```

С помощью квадратных скобок также указывается **индекс** в массиве, что позволяет получать доступ к элементам по их позициям:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine(vowels[1]); // e
```

Код приведет к выводу буквы “*е*”, поскольку массив индексируется, начиная с 0. Оператор цикла **for** можно использовать для прохода по всем элементам в массиве. Цикл **for** в следующем примере выполняется для целочисленных значений *i* от 0 до 4:

```
for (int i = 0; i < vowels.Length; i++)
    Console.Write(vowels[i]); // aeiou
```

Свойство **Length** массива возвращает количество элементов в массиве. После создания массива изменять его длину нельзя. Пространство имен **System.Collection** и вложенные в него пространства имен предоставляют такие высокоуровневые структуры данных, как массивы с динамически изменяемыми размерами и словари.

Выражение инициализации массива позволяет объявлять и заполнять массив в единственном операторе:

```
char[] vowels = new char[] {'a', 'e', 'i', 'o', 'u'};
```

или проще:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
```



Начиная с версии C# 12, вместо фигурных скобок можно использовать квадратные:

```
char[] vowels = ['a', 'e', 'i', 'o', 'u'];
```

Это называется **выражением коллекции**, и его преимущество заключается в том, что оно работает и при вызове методов:

```
Foo(['a', 'e', 'i', 'o', 'u']);
void Foo(char[] letters) { ... }
```

Выражения коллекций также допускаются для других типов коллекций, таких как списки и наборы (см. раздел “Инициализаторы и выражения коллекций” в главе 4).

Все массивы унаследованы от класса `System.Array`, который предоставляет общие службы для всех массивов. В состав его членов входят методы для получения и установки элементов независимо от типа массива; они описаны в разделе “Класс `Array`” главы 7.

Стандартная инициализация элементов

При создании массива всегда происходит инициализация его элементов стандартными значениями. Стандартное значение для типа представляет собой результат побитового обнуления памяти. Например, пусть создается массив целых чисел. Поскольку `int` — тип значения, выделится пространство под 1000 целочисленных значений в непрерывном блоке памяти. Стандартным значением для каждого элемента будет 0:

```
int[] a = new int[1000];
Console.WriteLine(a[123]); // 0
```

Типы значений или ссылочные типы

Значительное влияние на производительность оказывает то, какой тип имеют элементы массива — тип значения или ссылочный тип. Если элементы относятся к типу значения, то пространство под значение каждого элемента выделяется как часть массива:

```
Point[] a = new Point[1000];
int x = a[500].X; // 0
public struct Point { public int X, Y; }
```

Если бы типом `Point` был класс, тогда создание массива привело бы просто к выделению пространства под 1000 ссылок `null`:

```
Point[] a = new Point[1000];
int x = a[500].X; // Ошибка во время выполнения,
// исключение NullReferenceException
public class Point { public int X, Y; }
```

Чтобы устранить ошибку, после создания экземпляра массива потребуется явно создать 1000 экземпляров `Point`:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // Цикл для i от 0 до 999
    a[i] = new Point(); // Установить i-ый элемент массива
    // в новый экземпляр Point
```

Независимо от типа элементов массив *сам по себе* всегда является объектом ссылочного типа. Например, следующий оператор допустим:

```
int[] a = null;
```

Индексы и диапазоны

Индексы и диапазоны (появившиеся в C# 8) упрощают работу с элементами или порциями массива.



Индексы и диапазоны работают также с CLR-типами `Span<T>` и `ReadOnlySpan<T>` (см. главу 23).

Вы можете заставить работать с индексами и диапазонами также и собственные типы, определив индексатор типа `Index` или `Range` (см. раздел “Индексаторы” в главе 3).

Индексы

Индексы позволяют ссылаться на элементы относительно конца массива с применением операции `^`. Скажем, `^1` ссылается на последний элемент, `^2` — на предпоследний элемент и т.д.:

```
char[] vowels = new char[] {'a','e','i','o','u'};  
char lastElement = vowels [^1]; // 'u'  
char secondToLast = vowels [^2]; // 'o'
```

(`^0` равно длине массива, так что `vowels[^0]` приведет к генерации ошибки.)

Индексы в C# реализованы с помощью типа `Index`, а потому вы можете поступать так:

```
Index first = 0;  
Index last = ^1;  
char firstElement = vowels [first]; // 'a'  
char lastElement = vowels [last]; // 'u'
```

Диапазоны

Диапазоны позволяют “нарезать” массив посредством операции `..:`

```
char[] firstTwo = vowels [..2]; // 'a', 'e'  
char[] lastThree = vowels [2..]; // 'i', 'o', 'u'  
char[] middleOne = vowels [2..3]; // 'i'
```

Второе число в диапазоне является исключающим, поэтому `..2` возвращает элементы, находящиеся перед `vowels[2]`.

Вы также можете использовать символ `^` в диапазонах. Следующий диапазон возвращает последние два символа:

```
char[] lastTwo = vowels [^2..]; // 'o', 'u'
```

Диапазоны в C# реализуются с помощью типа `Range`, так что показанный ниже код допустим:

```
Range firstTwoRange = 0..2;  
char[] firstTwo = vowels [firstTwoRange]; // 'a', 'e'
```

Многомерные массивы

Многомерные массивы бывают двух видов: *прямоугольные* и *зубчатые*. Прямоугольный массив представляет *n*-мерный блок памяти, а зубчатый массив является массивом, содержащим массивы.

Прямоугольные массивы

Прямоугольные массивы объявляются с применением запятых для отделения каждого измерения друг от друга. Ниже приведено объявление прямоугольного двумерного массива с размерностью 3×3 :

```
int[,] matrix = new int[3,3];
```

Метод `GetLength` массива возвращает длину для заданного измерения (начиная с 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
    for (int j = 0; j < matrix.GetLength(1); j++)
        matrix[i,j] = i * 3 + j;
```

Прямоугольный массив может быть инициализирован явными значениями.

В следующем коде создается массив, идентичный массиву из предыдущего примера:

```
int[,] matrix = new int[,]
{
    {0,1,2},
    {3,4,5},
    {6,7,8}
};
```

Зубчатые массивы

Зубчатые массивы объявляются с использованием последовательно идущих пар квадратных скобок, которые представляют каждое измерение. Ниже показан пример объявления зубчатого двумерного массива с самым внешним измерением, составляющим 3:

```
int[][] matrix = new int[3][];
```



Обратите внимание на применение конструкции `new int[3][]`, а не `new int[] [3]`. Эрик Липперт написал великолепную статью с объяснениями, почему это так: <http://albahari.com/jagged>.

Внутренние измерения в объявлении не указываются, т.к. в отличие от прямоугольного массива каждый внутренний массив может иметь произвольную длину. Каждый внутренний массив неявно инициализируется значением `null`, а не пустым массивом. Каждый внутренний массив должен создаваться вручную:

```
for (int i = 0; i < matrix.Length; i++)
{
    matrix[i] = new int[3]; // Создать внутренний массив
    for (int j = 0; j < matrix[i].Length; j++)
        matrix[i][j] = i * 3 + j;
}
```

Зубчатый массив может быть инициализирован явными значениями. В следующем коде создается массив, который практически идентичен массиву из предыдущего примера, но имеет дополнительный элемент в конце:

```
int[][] matrix = new int[][] {
    new int[] {0,1,2},
    new int[] {3,4,5},
    new int[] {6,7,8,9}
};
```

Упрощенные выражения инициализации массивов

Существуют два способа сократить выражения инициализации массивов. Первый из них заключается в том, чтобы опустить операцию new и уточнители типов:

```
char[] vowels = {'a','e','i','o','u'};  
int[,] rectangularMatrix =  
{  
    {0,1,2},  
    {3,4,5},  
    {6,7,8}  
};  
int[][] jaggedMatrix =  
{  
    new int[] {0,1,2},  
    new int[] {3,4,5},  
    new int[] {6,7,8,9}  
};
```

(Начиная с версии C# 12, для одномерных массивов вместо фигурных скобок можно применять квадратные.)

Второй подход предусматривает использование ключевого слова var, которое сообщает компилятору о необходимости неявной типизации локальной переменной. Ниже приведены простые примеры:

```
var i = 3; // i неявно получает тип int  
var s = "sausage"; // s неявно получает тип string
```

Тот же самый принцип применим к массивам, причем его можно продвинуть на шаг дальше: опустить уточнитель типа после ключевого слова new и позволить компилятору самостоятельно вывести тип массива:

```
var vowels = new[] {'a','e','i','o','u'}; // Компилятор выведет тип char[]
```

Вот как его применять к многомерным массивам:

```
var rectMatrix = new [,] // rectMatrix неявно получает тип int[,]  
{  
    {0,1,2},  
    {3,4,5},  
    {6,7,8}  
};  
var jaggedMat = new int[][] // jaggedMat неявно получает тип int[][]  
{  
    new [] {0,1,2},  
    new [] {3,4,5},  
    new [] {6,7,8,9}  
};
```

Чтобы такой прием работал, элементы должны быть неявно преобразуемы в единственный тип (и хотя бы один элемент должен относиться к этому типу плюс должен существовать в точности один наилучший тип), как в следующем примере:

```
var x = new[] {1,10000000000}; // Все элементы преобразуемы в тип long
```

Проверка границ

Во время выполнения все обращения к индексам массивов проверяются на предмет выхода за допустимые границы. В случае указания недопустимого значения индекса генерируется исключение `IndexOutOfRangeException`:

```
int[] arr = new int[3];
arr[3] = 1; // Генерируется исключение IndexOutOfRangeException
```

Проверка границ в массивах необходима для обеспечения безопасности типов и упрощения отладки.



Обычно влияние на производительность проверки границ оказывается незначительным, и компилятор JIT способен проводить оптимизацию, такую как выяснение перед входом в цикл, будут ли все индексы безопасными, устранивая тем самым потребность в проверке на каждой итерации. Вдобавок в языке C# поддерживается “небезопасный” код, где можно явно пропускать проверку границ (см. раздел “Небезопасный код и указатели” в главе 4).

Переменные и параметры

Переменная представляет ячейку в памяти, которая содержит изменяемое значение. Переменная может быть *локальной переменной, параметром (передаваемым по значению, ref, out либо in), полем (экземпляра либо статическим)* или *элементом массива*.

Стек и куча

Стек и куча являются местами для хранения переменных. Стек и куча имеют существенно отличающуюся семантику времени жизни.

Стек

Стек представляет собой блок памяти для хранения локальных переменных и параметров. Стек логически расширяется и сужается при входе и выходе в метод или функцию. Взгляните на следующий метод (чтобы не отвлекать внимание, проверка входного аргумента не делается):

```
static int Factorial (int x)
{
    if (x == 0) return 1;
    return x * Factorial (x-1);
}
```

Метод `Factorial` является рекурсивным, т.е. вызывает сам себя. Каждый раз, когда происходит вход в метод, в стеке размещается экземпляр `int`, и каждый раз, когда метод завершается, экземпляр `int` освобождается.

Куча

Куча представляет собой блок памяти, где располагаются *объекты* (т.е. экземпляры ссылочного типа). Всякий раз, когда создается новый объект, он размещается в куче с возвращением ссылки на созданный объект. Во время выполнения программы куча начинает заполняться по мере создания новых объектов. В исполняющей среде предусмотрен сборщик мусора, который периодически освобождает объекты из кучи, поэтому программа не сталкивается с ситуацией нехватки памяти. Объект становится пригодным для освобождения, если на него не ссылается что-то, что само существует.

В приведенном ниже примере мы начинаем с создания объекта `StringBuilder`, на который ссылается переменная `ref1`, и выводим на экран его содержимое. Затем этот объект `StringBuilder` может быть немедленно обработан сборщиком мусора, т.к. впоследствии он нигде не задействован.

Далее мы создаем еще один объект `StringBuilder`, на который ссылается переменная `ref2`, и копируем ссылку в `ref3`. Хотя `ref2` в дальнейшем не применяется, переменная `ref3` поддерживает существование объекта `StringBuilder`, гарантируя тем самым, что он не будет подвергаться сборке мусора до тех пор, пока не мы закончим работу с `ref3`:

```
using System;
using System.Text;

StringBuilder ref1 = new StringBuilder ("object1");
Console.WriteLine (ref1);
//Объект StringBuilder, на который ссылается ref1, теперь пригоден для сборки мусора
StringBuilder ref2 = new StringBuilder ("object2");
StringBuilder ref3 = ref2;
// Объект StringBuilder, на который ссылается ref2, пока еще НЕ пригоден
// для сборки мусора.

Console.WriteLine (ref3); // object2
```

Экземпляры типов значений (и ссылки на объекты) хранятся там, где были объявлены соответствующие переменные. Если экземпляр был объявлен как поле внутри типа класса или как элемент массива, то такой экземпляр попадает в кучу.



В языке C# нельзя явно удалять объекты, как разрешено делать в C++. Объект без ссылок со временем будет уничтожен сборщиком мусора.

В куче также хранятся статические поля. В отличие от объектов, размещенных в куче (которые могут быть обработаны сборщиком мусора), они существуют до тех пор, пока не завершится процесс.

Определенное присваивание

В C# принудительно применяется политика определенного присваивания. На практике это означает, что за пределами контекста `unsafe` или контекста взаимодействия случайно получить доступ к неинициализированной памяти невозможно.

Определенное присваивание приводит к трем последствиям.

- Локальным переменным должны быть присвоены значения, прежде чем их можно будет читать.
- При вызове метода должны быть предоставлены аргументы функции (если только они не помечены как необязательные; см. раздел “Необязательные параметры” далее в главе).
- Все остальные переменные (такие как поля и элементы массивов) автоматически инициализируются исполняющей средой.

Например, следующий код приводит к ошибке на этапе компиляции:

```
int x;  
Console.WriteLine (x); // Ошибка на этапе компиляции
```

Поля и элементы массива автоматически инициализируются стандартными значениями для своих типов. Показанный ниже код выводит на экран 0, потому что элементам массива неявно присвоены их стандартные значения:

```
int[] ints = new int[2];  
Console.WriteLine (ints[0]); // 0
```

Следующий код выводит 0, т.к. полям (статическим или экземпляра) неявно присваиваются стандартные значения:

```
Console.WriteLine (Test.X); // 0  
class Test { public static int X; } // Поле
```

Стандартные значения

Экземпляры всех типов имеют стандартные значения. Стандартные значения для предопределенных типов являются результатом побитового обнуления памяти.

Тип	Стандартное значение
Ссыльчные типы (и типы значений, допускающие null)	null
Числовые и перечислимые типы	0
Тип char	'\0'
Тип bool	false

Получить стандартное значение для любого типа можно с помощью ключевого слова default:

```
Console.WriteLine (default (decimal)); // 0
```

Кроме того, когда тип может быть выведен, можно его не указывать:

```
decimal d = default;
```

Стандартное значение в специальном типе значения (т.е. struct) — это тоже самое, что и стандартные значения для всех полей, определенных в специальном типе.

Параметры

Метод может иметь последовательность параметров. Параметры определяют набор аргументов, которые должны быть предоставлены данному методу. В следующем примере метод Foo имеет единственный параметр по имени р типа int:

```
Foo (8); // 8 - аргумент  
static void Foo (int p) {...} // p - параметр
```

Управлять способом передачи параметров можно посредством модификаторов ref, in и out.

Модификатор параметра	Способ передачи	Когда переменная должна быть определено присвоена
Отсутствует	По значению	При входе
ref	По ссылке	При входе
in	По ссылке (только для чтения)	При входе
out	По ссылке	При выходе

Передача аргументов по значению

По умолчанию аргументы в C# передаются по значению, что общепризнанно является самым распространенным случаем. Другими словами, при передаче значения методу создается его копия:

```
int x = 8;  
Foo (x); // Создается копия x  
Console.WriteLine (x); // x по-прежнему будет иметь значение 8  
static void Foo (int p)  
{  
    p = p + 1; // Увеличить p на 1  
    Console.WriteLine (p); // Вывести значение p на экран  
}
```

Присваивание p нового значения не изменяет содержимое x, поскольку p и x находятся в разных ячейках памяти.

Передача по значению аргумента ссылочного типа приводит к копированию ссылки, но не объекта. В следующем примере метод Foo видит тот же самый объект StringBuilder, который был создан (sb), но имеет независимую ссылку на него. Другими словами, sb и fooSB являются отдельными друг от друга переменными, которые ссылаются на один и тот же объект StringBuilder:

```
StringBuilder sb = new StringBuilder();  
Foo (sb);  
Console.WriteLine (sb.ToString()); // test  
static void Foo (StringBuilder fooSB)  
{  
    fooSB.Append ("test");  
    fooSB = null;  
}
```

Из-за того, что `fooSB` — копия ссылки, установка ее в `null` не приводит к установке в `null` переменной `sb`. (Тем не менее, если параметр `fooSB` объявить и вызвать с модификатором `ref`, то `sb` станет равным `null`.)

Модификатор `ref`

Для передачи по ссылке в C# предусмотрен модификатор параметра `ref`. В приведенном далее примере `p` и `x` ссылаются на одну и ту же ячейку памяти:

```
int x = 8;
Foo (ref x);                                // Позволить Foo работать напрямую с x
Console.WriteLine (x);                      // x теперь имеет значение 9

static void Foo (ref int p)
{
    p = p + 1;                            // Увеличить p на 1
    Console.WriteLine (p);                // Вывести значение p на экран
}
```

Теперь присваивание `p` нового значения изменяет содержимое `x`. Обратите внимание, что модификатор `ref` должен быть указан как при определении, так и при вызове метода⁴. Такое требование делает очень ясным то, что происходит.

Модификатор `ref` критически важен при реализации метода обмена (в разделе “Обобщения” главы 3 будет показано, как реализовать метод обмена, работающий с любым типом):

```
string x = "Penn";
string y = "Teller";
Swap (ref x, ref y);
Console.WriteLine (x);                    // Teller
Console.WriteLine (y);                    // Penn

static void Swap (ref string a, ref string b)
{
    string temp = a;
    a = b;
    b = temp;
}
```



Параметр может быть передан по ссылке или по значению независимо от того, относится он к ссылочному типу или к типу значения.

Модификатор `out`

Аргумент `out` похож на аргумент `ref` за исключением следующих аспектов:

- он не нуждается в присваивании значения перед входом в функцию;
- ему должно быть присвоено значение перед выходом из функции.

⁴ Исключением из этого правила является вызов методов СОМ. Мы обсудим данную тему в главе 25.

Модификатор **out** чаще всего применяется для получения из метода нескольких возвращаемых значений, например:

```
string a, b;
Split ("Stevie Ray Vaughn", out a, out b);
Console.WriteLine (a);                                     // Stevie Ray
Console.WriteLine (b);                                     // Vaughn

void Split (string name, out string firstNames, out string lastName)
{
    int i = name.LastIndexOf (' ');
    firstNames = name.Substring (0, i);
    lastName = name.Substring (i + 1);
}
```

Подобно параметру **ref** параметр **out** передается по ссылке.

Переменные **out** и отбрасывание

Переменные можно объявлять на лету при вызове методов с параметрами **out**. Вот как можно заменить первые две строки из предыдущего примера:

```
Split ("Stevie Ray Vaughan", out string a, out string b);
```

Иногда при вызове методов с многочисленными параметрами **out** вы не заинтересованы в получении значений из всех параметров. В таких ситуациях можно с помощью символа подчеркивания “отбросить” те параметры, которые не представляют для вас интерес:

```
Split ("Stevie Ray Vaughan", out string a, out _);      // Отбросить второй
                                                               // параметр out
Console.WriteLine (a);
```

В данном случае компилятор трактует символ подчеркивания как специальный символ, называемый *отбрасыванием*. В одиночный вызов допускается включать множество символов отбрасывания. Предполагая, что метод **SomeBigMethod** был определен с семью параметрами **out**, вот как проигнорировать все кроме четвертого:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
```

В целях обратной совместимости данное языковое средство не вступит в силу, если в области видимости находится реальная переменная с именем в виде символа подчеркивания:

```
string _;
Split ("Stevie Ray Vaughan", out string a, out _);
Console.WriteLine (_);                                    // Vaughan
```

Последствия передачи по ссылке

Когда вы передаете аргумент по ссылке, то устанавливаете псевдоним для ячейки памяти, в которой находится существующая переменная, а не создаете новую ячейку. В следующем примере переменные **x** и **y** представляют один и тот же экземпляр:

```

class Test
{
    static int x;

    static void Main() { Foo (out x); }

    static void Foo (out int y)
    {
        Console.WriteLine (x);      // x имеет значение 0
        y = 1;                    // Изменить значение y
        Console.WriteLine (x);      // x имеет значение 1
    }
}

```

Модификатор `in`

Параметр `in` похож на параметр `ref` за исключением того, что значение аргумента не может быть модифицировано в методе (его изменение приведет к генерации ошибки на этапе компиляции). Модификатор `in` наиболее полезен при передаче методу крупного типа значения, потому что он позволяет компилятору избежать накладных расходов, связанных с копированием аргумента перед его передачей, сохраняя при этом защиту первоначального значения от модификации.

Перегрузка допускается только при наличии модификатора `in`:

```

void Foo ( SomeBigStruct a) { ... }
void Foo (in SomeBigStruct a) { ... }

```

Для вызова второй перегруженной версии в вызывающем коде должен присутствовать модификатор `in`:

```

SomeBigStruct x = ...;
Foo (x);           // Вызывается первая перегруженная версия
Foo (in x);        // Вызывается вторая перегруженная версия

```

Когда неоднозначность отсутствует:

```
void Bar (in SomeBigStruct a) { ... }
```

то указывать модификатор `in` в вызывающем коде необязательно:

```

Bar (x);           // Нормально (вызывается перегруженная версия с in)
Bar (in x);        // Нормально (вызывается перегруженная версия с in)

```

Чтобы сделать приведенный пример содержательным, `SomeBigStruct` следовало бы определить как структуру (см. раздел “Структуры” в главе 3).

Модификатор `params`

Модификатор `params`, примененный к последнему параметру метода, позволяет методу принимать любое количество аргументов определенного типа. Тип параметра должен быть объявлен как (одномерный) массив, например:

```

int total = Sum (1, 2, 3, 4);
Console.WriteLine (total); // 10

// Вызов Sum выше в коде эквивалентен:
int total2 = Sum (new int[] { 1, 2, 3, 4 });

```

```
int Sum (params int[] ints)
{
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
        sum += ints [i]; // Увеличить sum на ints[i]
    return sum;
}
```

Если в позиции `params` аргументы отсутствуют, тогда создается массив нулевой длины.

Аргумент `params` можно также предоставить как обычный массив. Первая строка кода в `Main` семантически эквивалентна следующей строке:

```
int total = Sum (new int[] { 1, 2, 3, 4 });
```

Необязательные параметры

В методах, конструкторах и индексаторах (глава 3) можно объявлять **необязательные параметры**. Параметр является необязательным, если в его объявлении указано **стандартное значение**:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

При вызове метода необязательные параметры могут быть опущены:

```
Foo(); // 23
```

Необязательному параметру `x` в действительности *передается стандартный аргумент* со значением 23 — компилятор встраивает это значение в скомпилированный код на *вызывающей* стороне. Показанный выше вызов `Foo` семантически эквивалентен следующему вызову:

```
Foo (23);
```

поскольку компилятор просто подставляет стандартное значение необязательного параметра там, где он используется.



Добавление необязательного параметра к открытому методу, который вызывается из другой сборки, требует перекомпиляции обеих сборок — как и в случае, если бы параметр был обязательным.

Стандартное значение необязательного параметра должно быть указано в виде константного выражения, вызова конструктора без параметров для типа значения или стандартного выражения. Необязательные параметры не могут быть помечены как `ref` или `out`.

Обязательные параметры должны находиться *перед* необязательными параметрами в объявлении метода и его вызове (исключением являются аргументы `params`, которые всегда располагаются в конце). В следующем примере параметру `x` передается явное значение 1, а параметру `y` — стандартное значение 0:

```
Foo (1); // 1, 0
void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }
```

Чтобы сделать обратное (передать стандартное значение для `x` и явное значение для `y`), потребуется скомбинировать необязательные параметры с *именованными аргументами*.

Именованные аргументы

Вместо распознавания аргумента по позиции его можно идентифицировать по имени:

```
Foo (x:1, y:2); // 1, 2
void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }
```

Именованные аргументы могут указываться в любом порядке. Следующие вызовы Foo семантически идентичны:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```



Тонкое отличие состоит в том, что выражения в аргументах вычисляются согласно порядку, в котором они появляются на вызывающей стороне. В общем случае это актуально только для взаимозависимых выражений с побочными эффектами, как в следующем коде, который выводит на экран 0, 1:

```
int a = 0;
Foo (y: ++a, x: --a); // Выражение ++a вычисляется первым
```

Разумеется, на практике вы определенно должны избегать подобного стиля кодирования!

Именованные и позиционные аргументы можно смешивать:

```
Foo (1, y:2);
```

Однако существует одно ограничение: позиционные аргументы должны находиться перед именованными аргументами, если только они не используются в корректных позициях. Таким образом, мы могли бы вызвать Foo следующим образом:

Foo (x:1, 2); // Нормально. Аргументы находятся в объявленных позициях
но не так, как показано ниже:

```
Foo (y:2, 1); // Ошибка на этапе компиляции. y находится не в первой позиции
```

Именованные аргументы особенно удобны в сочетании с необязательными параметрами. Например, взгляните на следующий метод:

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) { ... }
```

Его можно вызвать, предоставив только значение для d:

```
Bar (d:3);
```

Как будет подробно обсуждаться в главе 24, это очень удобно при работе с API-интерфейсами COM.

Локальные ссылочные переменные

В версии C# 7 появилось довольно-таки экзотическое средство, позволяющее определять локальную переменную, которая ссылается на элемент в массиве или на поле в объекте:

```
int[] numbers = { 0, 1, 2, 3, 4 };
ref int numRef = ref numbers [2];
```

В приведенном примере numRef является ссылкой на numbers[2]. Модификация numRef приводит к модификации элемента массива:

```
numRef *= 10;
Console.WriteLine (numRef);           // 20
Console.WriteLine (numbers [2]);      // 20
```

В качестве цели ссылочной локальной переменной должен указываться элемент массива, поле или обычная локальная переменная; целью не может быть *свойство* (глава 3). Локальные ссылочные переменные предназначены для специализированных сценариев микрооптимизации и обычно применяются в сочетании с *возвращаемыми ссылочными значениями*.

Возвращаемые ссылочные значения



Типы `Span<T>` и `ReadOnlySpan<T>`, которые будут описаны в главе 23, используют возвращаемые ссылочные значения для реализации высокоэффективного индексатора. Помимо сценариев подобного рода возвращаемые ссылочные значения обычно не применяются; вы можете считать их средством микрооптимизации.

Ссылочную локальную переменную можно возвращать из метода. Результат называется *возвращаемым ссылочным значением*:

```
class Program
{
    static string x = "Old Value";

    static ref string GetX() => ref x;    // Этот метод возвращает
                                            // ссылочное значение

    static void Main()
    {
        ref string xRef = ref GetX();       // Присвоить результат ссылочной
                                            // локальной переменной
        xRef = "New Value";
        Console.WriteLine (x);             // Выводит New Value
    }
}
```

Если вы опустите модификатор `ref` на вызывающей стороне, тогда будет возвращаться обычное значение:

```
string localX = GetX();                // Допустимо: localX - обыкновенная,
                                            // не ссылочная переменная
```

Вы можете использовать возвращаемые ссылочные значения при определении свойства или индексатора:

```
static ref string Prop => ref x;
```

Такое свойство неявно допускает запись, несмотря на отсутствие средства доступа `set`:

```
Prop = "New Value";
```

Воспрепятствовать модификации можно за счет применения `ref readonly`:

```
static ref readonly string Prop => ref x;
```

Модификатор `ref readonly` предотвращает модификацию, одновременно обеспечивая выигрыш в производительности при возврате по ссылке. В данном случае выигрыш будет небольшим, потому что `x` имеет тип `string` (ссылочный тип): независимо от длины строки единственной неэффективностью, которой мы можем избежать, является копирование одиночной 32- или 64-битной ссылки. Реальный выигрыш может быть получен со специальными типами значений (см. раздел “Структуры” в главе 3), но только если структура помечена как `readonly` (иначе компилятор будет выполнять защитное копирование).

Определять явное средство доступа `set` для свойства или индексатора созвращаемым ссылочным значением не разрешено.

Объявление неявно типизированных локальных переменных с помощью `var`

Часто случается так, что переменная объявляется и инициализируется за один шаг. Если компилятор способен вывести тип из инициализирующего выражения, то на месте объявления типа можно использовать ключевое слово `var`, например:

```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Приведенный код в точности эквивалентен следующему коду:

```
string x = "hello";
System.Text.StringBuilder y = new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Из-за такой прямой эквивалентности неявно типизированные переменные являются статически типизированными. Скажем, показанный ниже код вызовет ошибку на этапе компиляции:

```
var x = 5;
x = "hello"; // Ошибка на этапе компиляции; x относится к типу int
```



Применение `var` может ухудшить читабельность кода в случае, если вы не можете вывести тип, просто взглянув на объявление переменной. Ниже приведен пример:

```
Random r = new Random();
var x = r.Next();
```

Какой тип имеет переменная `x`?

В разделе “Анонимные типы” главы 4 мы опишем сценарий, в котором использование ключевого слова `var` обязательно.

Выражения new целевого типа

Начиная с C# 9, еще один способ сокращения лексического повторения предлагаю выражения new целевого типа:

```
System.Text.StringBuilder sb1 = new();
System.Text.StringBuilder sb2 = new ("Test");
```

Код в точности эквивалентен следующему коду:

```
System.Text.StringBuilder sb1 = new System.Text.StringBuilder();
System.Text.StringBuilder sb2 = new System.Text.StringBuilder ("Test");
```

Принцип заключается в том, что вы можете обращаться к new, не указывая имя типа, если компьютер способен однозначно вывести его. Выражения new целевого типа особенно удобны, когда объявление и инициализация переменных находятся в разных частях кода. Распространенным примером может служить инициализация поля в конструкторе:

```
class Foo
{
    System.Text.StringBuilder sb;
    public Foo (string initialValue)
    {
        sb = new (initialValue);
    }
}
```

Выражения new целевого типа также удобны в следующем сценарии:

```
MyMethod (new ("test"));
void MyMethod (System.Text.StringBuilder sb) { ... }
```

Выражения и операции

Выражение по существу обозначает значение. Простейшими видами выражений являются константы и переменные. Выражения могут видоизменяться и комбинироваться с применением операций. *Операция* принимает один или большее количество входных *операндов*, формируя новое выражение.

Вот пример *константного выражения*:

12

Посредством операции * можно скомбинировать два операнда (литеральные выражения 12 и 30):

12 * 30

Мы можем строить сложные выражения, потому что операнд сам по себе может быть выражением, как операнд (12 * 30) в следующем примере:

1 + (12 * 30)

Операции в C# могут быть классифицированы как *унарные*, *бинарные* или *тернарные* в зависимости от количества operandов, с которыми они работают (один, два или три). Бинарные операции всегда используют *инфиксную* форму, когда операция помещается между двумя operandами.

Первичные выражения

Первичные выражения включают выражения, сформированные из операций, которые являются неотъемлемой частью самого языка. Ниже показан пример:

```
Math.Log (1)
```

Выражение здесь состоит из двух первичных выражений. Первое выражение осуществляет поиск члена (посредством операции .), а второе — вызов метода (с помощью операции ()).

Пустые выражения

Пустое выражение — это выражение, которое не имеет значения; например:

```
Console.WriteLine (1)
```

Поскольку пустое выражение не имеет значения, его нельзя применять в качестве операнда при построении более сложных выражений:

```
1 + Console.WriteLine (1) // Ошибка на этапе компиляции
```

Выражения присваивания

Выражение присваивания использует операцию = для присваивания переменной результата вычисления другого выражения, например:

```
x = x * 5
```

Выражение присваивания — это не пустое выражение. Оно заключает в себе значение, которое было присвоено, и потому может встраиваться в другое выражение. В следующем примере выражение присваивает значение 2 переменной x и 10 переменной y:

```
y = 5 * (x = 2)
```

Такой стиль выражения может применяться для инициализации нескольких значений:

```
a = b = c = d = 0
```

Составные операции присваивания являются синтаксическим сокращением, которое комбинирует присваивание с другой операцией:

<pre>x *= 2</pre>	<i>// Эквивалентно x = x * 2</i>
<pre>x <= 1</pre>	<i>// Эквивалентно x = x << 1</i>

(Тонкое исключение из указанного правила касается *событий*, которые рассматриваются в главе 4: операции += и -= в них трактуются особым образом и отображаются на средства доступа add и remove события.)

Приоритеты и ассоциативность операций

Когда выражение содержит несколько операций, порядок их вычисления определяется *приоритетами* и *ассоциативностью*. Операции с более высоким приоритетом выполняются перед операциями, приоритет которых ниже. Если операции имеют одинаковый приоритет, то порядок их выполнения определяется ассоциативностью.

Приоритеты операций

Приведенное ниже выражение:

`1 + 2 * 3`

вычисляется следующим образом, т.к. операция `*` имеет больший приоритет, чем `+`:

`1 + (2 * 3)`

Левоассоциативные операции

Бинарные операции (кроме операции присваивания, лямбда-операции и операции объединения с `null`) являются *левоассоциативными*; другими словами, они вычисляются слева направо. Например, выражение:

`8 / 4 / 2`

вычисляется так:

`(8 / 4) / 2 // 1`

Чтобы изменить фактический порядок вычисления, можно расставить скобки:

`8 / (4 / 2) // 4`

Правоассоциативные операции

Операции присваивания, лямбда-операция, операция объединения с `null` и условная операция являются *правоассоциативными*; другими словами, они вычисляются справа налево.

Правая ассоциативность делает возможной успешную компиляцию множественного присваивания вроде показанного ниже:

`x = y = 3;`

Здесь значение 3 сначала присваивается переменной `y`, после чего результат этого выражения (3) присваивается переменной `x`.

Таблица операций

В табл. 2.3 перечислены операции C# в порядке их приоритетов. Операции в одной и той же категории имеют одинаковые приоритеты. Операции, которые могут быть перегружены пользователем, объясняются в разделе “Перегрузка операций” главы 4.

Операции для работы со значениями `null`

В языке C# предлагаются три операции, которые предназначены для упрощения работы со значениями `null`: *операция объединения с null* (`null-coalescing operator`), *операция присваивания с объединением с null* (`null-coalescing assignment operator`) и *null-условная операция* (`null-conditional operator`).

Таблица 2.3. Операции C# (с категоризацией в порядке приоритетов)

Категория	Символ операции	Название операции	Пример	Возможность перегрузки пользователем
Первичные	.	Доступ к члену	x.y	Нет
	? . и ? []	null-условная	x?.y или x?[0]	Нет
	! (постфиксная)	null-терпимая	x!.y или x![0]	Нет
	-> (небезопасная)	Указатель на структуру	x->y	Нет
	()	Вызов функции	x()	Нет
	[]	Массив/индекс	a[x]	Через индексатор
	++	Постфиксная форма инкремента	x++	Да
	--	Постфиксная форма декремента	x--	Да
	new	Создание экземпляра	new Foo()	Нет
	stackalloc	Выделение памяти в стеке	stackalloc(10)	Нет
	typeof	Получение типа по идентификатору	typeof(int)	Нет
	nameof	Получение имени идентификатора	nameof(x)	Нет
	checked	Включение проверки целочисленного переполнения	checked(x)	Нет
	unchecked	Отключение проверки целочисленного переполнения	unchecked(x)	Нет
Унарные	default	Стандартное значение	default(char)	Нет
	await	Ожидание	await myTask	Нет
	sizeof	Получение размера структуры	sizeof(int)	Нет
	+	Положительное значение	+x	Да
	-	Отрицательное значение	-x	Да
	!	НЕ	!x	Да
	~	Побитовое дополнение	~x	Да
	++	Префиксная форма инкремента	++x	Да

Категория	Символ операции	Название операции	Пример	Возможность перегрузки пользователем
Диапазона	--	Префиксная форма декремента	--x	Да
	()	Приведение	(int)x	Нет
	^	Индекс с конца	array[^1]	Нет
	* (небезопасная)	Значение по адресу	*x	Нет
	& (небезопасная)	Адрес значения	&x	Нет
	..	Диапазон индексов	x..y	Нет
	..^		x..^y	
	switch и with	Выражение switch	num switch { 1 => true, _ => false }	Нет
	with	Выражение with	rec with { x = 123 }	Нет
	*	Умножение	x * y	Да
Мультипликативные	/	Деление	x / y	Да
	%	Остаток от деления	x % y	Да
	+	Сложение	x + y	Да
Аддитивные	-	Вычитание	x - y	Да
	<<	Сдвиг влево	x << 1	Да
	>>	Сдвиг вправо	x >> 1	Да
Сдвига	>>>	Беззнаковый сдвиг вправо	x >>> 1	Да
	<	Меньше	x < y	Да
	>	Больше	x > y	Да
	<=	Меньше или равно	x <= y	Да
	>=	Больше или равно	x >= y	Да
	is	Принадлежность к типу или его подклассу	x is y	Нет
	as	Преобразование типа	x as y	Нет
Эквивалентности	==	Равно	x == y	Да
	!=	Не равно	x != y	Да
Поразрядное И	&	И	x & y	Да

Категория	Символ операции	Название операции	Пример	Возможность перегрузки пользователем
Поразрядное исключающее ИЛИ	<code>^</code>	Исключающее ИЛИ	<code>x ^ y</code>	Да
Поразрядное ИЛИ	<code> </code>	ИЛИ	<code>x y</code>	Да
Условное И	<code>&&</code>	Условное И	<code>x && y</code>	Через &
Условное ИЛИ	<code> </code>	Условное ИЛИ	<code>x y</code>	Через
Объединение с null	<code>??</code>	Объединение с null	<code>x ?? y</code>	Нет
Условная	<code>?:</code>	Условная	<code>isTrue ? thenThis : elseThis</code>	Нет
Присваивания и лямбда	<code>=</code>	Присваивание	<code>x = y</code>	Нет
	<code>*=</code>	Умножение с присваиванием	<code>x *= 2</code>	Через *
	<code>/=</code>	Деление с присваиванием	<code>x /= 2</code>	Через /
	<code>%=</code>	Остаток от деления с присваиванием	<code>x %= 2</code>	
	<code>+=</code>	Сложение с присваиванием	<code>x += 2</code>	Через +
	<code>-=</code>	Вычитание с присваиванием	<code>x -= 2</code>	Через -
	<code><<=</code>	Сдвиг влево с присваиванием	<code>x <<= 2</code>	Через <<
	<code>>>=</code>	Сдвиг вправо с присваиванием	<code>x >>= 2</code>	Через >>
	<code>>>>=</code>	Беззнаковый сдвиг вправо с присваиванием	<code>x >>>= 2</code>	Через >>>
	<code>&=</code>	Операция И с присваиванием	<code>x &= 2</code>	Через &
	<code>^=</code>	Операция исключающего ИЛИ с присваиванием	<code>x ^= 2</code>	Через ^
	<code> =</code>	Операция ИЛИ с присваиванием	<code>x = 2</code>	Через
	<code>??=</code>	Присваивание с объединением с null	<code>x ??= 0</code>	Нет
	<code>=></code>	Лямбда-операция	<code>x => x + 1</code>	Нет

Операция объединения с null

Операция объединения с null обозначается как `??`. Она выполняется следующим образом: если операнд слева не равен null, тогда возвращается его значение, а иначе возвращается другое значение. Например:

```
string s1 = null;
string s2 = s1 ?? "nothing"; // Переменная s2 получает значение "nothing"
```

Если левостороннее выражение не равно null, то правостороннее выражение никогда не вычисляется. Операция объединения с null также работает с типами, допускающими null (см. раздел “Типы значений, допускающие null” в главе 4).

Операция присваивания с объединением с null

Операция `??=` (появившаяся в версии C# 8) называется *операцией присваивания с объединением с null*. Она выполняется так: если операнд слева равен null, тогда правый операнд присваивается левому операнду. Взгляните на следующий код:

```
myVariable ??= someDefault;
```

Он эквивалентен такому коду:

```
if (myVariable == null) myVariable = someDefault;
```

Операция `??=` особенно удобна при реализации лениво вычисляемых свойств. Мы раскроем эту тему в разделе “Вычисляемые поля и ленивая оценка” главы 4.

null-условная операция

Операция `?.` называется *null-условной операцией* (или *элвис-операцией*). Она позволяет вызывать методы и получать доступ к членам подобно стандартной операции точки, но с той разницей, что если находящийся слева операнд равен null, то результатом выражения будет null без генерации исключения `NullReferenceException`:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString(); // Ошибка не возникает; взамен s получает
                           // значение null
```

Последняя строка кода эквивалентна следующему коду:

```
string s = (sb == null ? null : sb.ToString());
```

Выражения с null-условной операцией работают также с индексаторами:

```
string[] words = null;
string word = words?[1]; // Переменная word получает значение null
```

Встретив значение null, элвис-операция прекращает вычисление оставшейся части выражения. В приведенном далее примере переменная `s` получает значение null, несмотря на наличие стандартной операции точки между вызовами `ToString()` и `ToUpper()`:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper(); // Переменная s получает значение null
// и ошибка не возникает
```

Многократное использование элвис-операции необходимо, только если находящийся непосредственно слева операнд может быть равен null. Следующее выражение надежно работает в ситуациях, когда и x, и x.y могут быть равны null:

```
x?.y?.z
```

Оно эквивалентно такому выражению (за исключением того, что x.y вычисляется только один раз):

```
x == null ? null
: (x.y == null ? null : x.y.z)
```

Окончательное выражение должно иметь возможность принимать значение null. Показанный ниже код не является допустимым, потому что переменная типа int не может принимать значение null:

```
System.Text.StringBuilder sb = null;
int length = sb?.ToString().Length; // Не допускается: переменная int
// не может принимать значение null
```

Исправить положение можно за счет применения типа значения, допускающего null (см. раздел “Типы значений, допускающие null” в главе 4). На тот случай, если вы уже знакомы с типами значений, допускающими null, то вот как выглядит код:

```
int? length = sb?.ToString().Length; // Нормально: переменная int?
// может принимать значение null
```

null-условную операцию можно также использовать для вызова метода void:

```
someObject?.SomeVoidMethod();
```

Если переменная someObject равна null, тогда такой вызов становится “отсутствием операции” вместо того, чтобы приводить к генерации исключения NullReferenceException.

null-условная операция может применяться с часто используемыми членами типов, которые будут описаны в главе 3, в том числе с *методами, полями, свойствами и индексаторами*. Она также хорошо сочетается с операцией объединения с null:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "nothing"; // Переменная s получает
// значение "nothing"
```

Операторы

Функции состоят из операторов, которые выполняются последовательно в порядке их появления внутри программы. Блок *операторов* — это последовательность операторов, находящихся между фигурными скобками {}).

Операторы объявления

Оператор объявления переменных объявляет новую переменную и дополнительно способен инициализировать ее посредством выражения. Можно объявлять несколько переменных одного и того же типа, указывая их в списке с запятой в качестве разделителя:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

Объявление константы похоже на объявление переменной за исключением того, что после объявления константа не может быть изменена, а объявление обязательно должно сопровождаться инициализацией (см. раздел “Константы” в главе 3):

```
const double c = 2.99792458E08;
c += 10; // Ошибка на этапе компиляции
```

Локальные переменные

Областью видимости локальной переменной или локальной константы является текущий блок. Объявлять еще одну локальную переменную с тем же самым именем в текущем блоке или в любых вложенных блоках не разрешено:

```
int x;
{
    int y;
    int x; // Ошибка - переменная x уже определена
}
{
    int y; // Нормально - переменная y не находится в области видимости
}
Console.WriteLine(y); //Ошибка-переменная y находится за пределами области видимости
```



Область видимости переменной распространяется в *обоих направлениях* на всем протяжении ее блока кода. Это означает, что даже если в приведенном примере перенести первоначальное объявление `x` в конец метода, то будет получена та же ошибка. Поведение отличается от языка C++ и в чем-то необычно, учитывая недопустимость ссылки на переменную или константу до ее объявления.

Операторы выражений

Операторы выражений представляют собой выражения, которые также являются допустимыми операторами. Оператор выражения должен либо изменять состояние, либо вызывать что-то, что может изменять состояние. Изменение состояния по существу означает изменение переменной. Ниже перечислены возможные операторы выражений:

- выражения присваивания (включая выражения инкремента и декремента);
- выражения вызова методов (`void` и не `void`);
- выражения создания объектов.

Рассмотрим несколько примеров:

```
// Объявить переменные с помощью операторов объявления:  
string s;  
int x, y;  
System.Text.StringBuilder sb;  
// Операторы выражений  
x = 1 + 2;                                // Выражение присваивания  
x++;                                         // Выражение инкремента  
y = Math.Max (x, 5);                        // Выражение присваивания  
Console.WriteLine (y);                      // Выражение вызова метода  
sb = new StringBuilder();                   // Выражение присваивания  
new StringBuilder();                         // Выражение создания объекта
```

При вызове конструктора или метода, который возвращает значение, вы не обязаны использовать результат. Тем не менее, если этот конструктор или метод не изменяет состояние, то такой оператор совершенно бесполезен:

```
new StringBuilder();                         // Допустим, но бесполезен  
new string ('c', 3);                       // Допустим, но бесполезен  
x.Equals (y);                            // Допустим, но бесполезен
```

Операторы выбора

В C# имеются следующие механизмы для условного управления потоком выполнения программы:

- операторы выбора (`if`, `switch`);
- условная операция (`? :`);
- операторы цикла (`while`, `do..while`, `for`, `foreach`).

В текущем разделе рассматриваются две простейшие конструкции: операторы `if` и `switch`.

Оператор `if`

Оператор `if` выполняет некоторый оператор, если вычисление выражения `bool` в результате дает `true`:

```
if (5 < 2 * 3)  
    Console.WriteLine ("true");                // Выводит true
```

В качестве оператора может выступать блок кода:

```
if (5 < 2 * 3)  
{  
    Console.WriteLine ("true");  
    Console.WriteLine ("Let's move on!");  
}
```

Конструкция `else`

Оператор `if` может быть дополнительно снабжен конструкцией `else`:

```
if (2 + 2 == 5)  
    Console.WriteLine ("Does not compute");    // Не вычисляется  
else  
    Console.WriteLine ("False");                // Выводит False
```

Внутрь конструкции `else` можно помещать другой оператор `if`:

```
if (2 + 2 == 5)
    Console.WriteLine ("Does not compute"); // Выводит Does not compute
                                                // (Не вычисляется)
else
    if (2 + 2 == 4)
        Console.WriteLine ("Computes"); // Выводит Computes (Вычисляется)
```

Изменение потока выполнения с помощью фигурных скобок

Конструкция `else` всегда применяется к непосредственно предшествующему оператору `if` в блоке операторов:

```
if (true)
    if (false)
        Console.WriteLine();
    else
        Console.WriteLine ("executes"); // Выводит executes (выполняется)
```

Код семантически идентичен такому коду:

```
if (true)
{
    if (false)
        Console.WriteLine();
    else
        Console.WriteLine ("executes");
}
```

Переместив фигурные скобки, поток выполнения можно изменить:

```
if (true)
{
    if (false)
        Console.WriteLine();
}
else
    Console.WriteLine ("does not execute"); // Выводит does not execute
                                                // (не выполняется)
```

С помощью фигурных скобок вы явно заявляете о своих намерениях. Фигурные скобки могут улучшить читабельность вложенных операторов `if`, даже когда они не требуются компилятором. Важным исключением является следующий шаблон:

```
void TellMeWhatICanDo (int age)
{
    if (age >= 35)
        Console.WriteLine ("You can be president!"); // Вы можете стать
                                                       // президентом!
    else if (age >= 21)
        Console.WriteLine ("You can drink!"); // Вы можете выпивать!
    else if (age >= 18)
        Console.WriteLine ("You can vote!"); // Вы можете голосовать!
    else
        Console.WriteLine ("You can wait!"); // Вы можете лишь ждать!
}
```

Здесь операторы `if` и `else` были организованы так, чтобы сымитировать конструкцию “`elseif`” из других языков (и директиву препроцессора `#elif` в C#). Средство автоматического форматирования Visual Studio распознает такой шаблон и предохраняет отступы. Однако семантически каждый оператор `if`, следующий за `else`, функционально вложен внутрь конструкции `else`.

Оператор `switch`

Операторы `switch` позволяют реализовать ветвление потока выполнения программы на основе выбора из возможных значений, которые переменная способна принимать. Операторы `switch` могут дать в результате более ясный код, чем множество операторов `if`, поскольку они требуют только однократного вычисления выражения:

```
static void ShowCard(int cardNumber)
{
    switch (cardNumber)
    {
        case 13:
            Console.WriteLine ("King");           // Король
            break;
        case 12:
            Console.WriteLine ("Queen");         // Дама
            break;
        case 11:
            Console.WriteLine ("Jack");          // Валет
            break;
        case -1:           // Джокер соответствует -1
            goto case 12; // В этой игре джокер подсчитывается как дама
        default:          // Выполняется для любого другого значения cardNumber
            Console.WriteLine (cardNumber);
            break;
    }
}
```

В приведенном примере демонстрируется самый распространенный сценарий, при котором осуществляется переключение по *константам*. При указании константы вы ограничены встроенными числовыми типами, а также типами `bool`, `char`, `string` и `enum`.

В конце каждой конструкции `case` посредством одного из операторов перехода необходимо явно указывать, куда управление должно передаваться дальше (если только вы не хотите получить сквозное выполнение). Ниже перечислены возможные варианты:

- `break` (переход в конец оператора `switch`);
- `goto case x` (переход на другую конструкцию `case`);
- `goto default` (переход на конструкцию `default`);
- любой другой оператор перехода, а именно — `return`, `throw`, `continue` или `goto` метка.

Когда для нескольких значений должен выполняться тот же самый код, конструкции `case` можно записывать последовательно:

```
switch (cardNumber)
{
    case 13:
    case 12:
    case 11:
        Console.WriteLine ("Face card"); // Фигурная карта
        break;
    default:
        Console.WriteLine ("Plain card"); // Нефигурная карта
        break;
}
```

Такая особенность оператора `switch` может иметь решающее значение в плане обеспечения более ясного кода, чем в случае множества операторов `if-else`.

Переключение по типам



Переключение по типу является особым случаем переключения по *шаблону*. В последних версиях C# появилось несколько других шаблонов; полное обсуждение ищите в разделе “Шаблоны” главы 4.

Можно также переключаться по *типам* (начиная с версии C# 7):

```
TellMeTheType (12);
TellMeTheType ("hello");
TellMeTheType (true);
void TellMeTheType (object x) // object допускает любой тип
{
    switch (x)
    {
        case int i:
            Console.WriteLine ("It's an int!"); // Это не целочисленное значение!
            Console.WriteLine ($"The square of {i} is {i * i}"); // Вывод квадрата
            break;
        case string s:
            Console.WriteLine ("It's a string!"); // Это не строка!
            Console.WriteLine ($"The length of {s} is {s.Length}"); // Вывод
            // длины строки
            break;
        case DateTime:
            Console.WriteLine ("It's a DateTime"); // Значение типа DateTime
            break;
        default:
            Console.WriteLine ("I don't know what x is"); // Неизвестное значение
            break;
    }
}
```

(Тип `object` воспринимает переменную любого типа; мы подробно обсудим это в разделах “Наследование” и “Тип `object`” главы 3.) В каждой конструкции `case` указываются тип для сопоставления и переменная, которой нужно присвоить типизированное значение в случае совпадения (“шаблонная” переменная). В отличие от констант никаких ограничений на используемые типы не налагается.

Конструкцию `case` можно снабдить ключевым словом `when`:

```
switch (x)
{
    case bool b when b == true:      // Выполняется, только если b равно true
        Console.WriteLine ("True!");
        break;
    case bool b:
        Console.WriteLine ("False!");
        break;
}
```

При переключении по типам порядок следования конструкций `case` может быть важным (в отличие от случая с переключением по константам). Если поменять местами две конструкции `case`, то рассмотренный пример давал бы другой результат (на самом деле он даже не скомпилируется, поскольку компилятор определит, что вторая конструкция `case` недостижима). Исключением из данного правила является конструкция `default`, которая всегда выполняется последней вне зависимости от того, где находится.

Можно указывать друг за другом несколько конструкций `case`. Вызов `Console.WriteLine` в показанном ниже коде будет выполняться для любого значения с плавающей точкой, которое больше 1000:

```
switch (x)
{
    case float f when f > 1000:
    case double d when d > 1000:
    case decimal m when m > 1000:
        Console.WriteLine ("We can refer to x here but not f or d or m");
        // Мы можем здесь ссылаться на x, но не на f или d или m
        break;
}
```

В приведенном примере компилятор разрешает употреблять шаблонные переменные `f`, `d` и `m` только в конструкциях `when`. При вызове `Console.WriteLine` неизвестно, какая из трех переменных будет присвоена, а потому компилятор выносит их все за пределы области видимости.

В рамках одного оператора `switch` константы и шаблоны можно смешивать и сочетать. Вдобавок можно переключаться по значению `null`:

```
case null:
    Console.WriteLine ("Nothing here");
    break;
```

Выражения `switch`

Начиная с версии C# 8, конструкцию `switch` можно использовать в контексте выражения. В следующем коде приведен пример, в котором предполагается, что `cardName` имеет тип `int`:

```
string cardName = cardNumber switch
{
    13 => "King",      // Король
    12 => "Queen",     // Дама
    11 => "Jack",      // Валет
    _ => "Pip card"   // Нефигурная карта; эквивалентно default
};
```

Обратите внимание на то, что ключевое слово `switch` находится *после* имени переменной, и конструкции `case` являются выражениями (которые заканчиваются запятыми), а не операторами. Выражения `switch` более компактны, чем эквивалентные им операторы `switch`, и вы можете использовать их в запросах LINQ (см. главу 8).

Если вы опустите выражение по умолчанию (`_`) и `switch` не обнаружит соответствия, тогда сгенерируется исключение.

Вы также можете переключаться по множеству значений (шаблон кортежа):

```
int cardNumber = 12;
string suit = "spades";

string cardName = (cardNumber, suit) switch
{
    (13, "spades") => "King of spades",
    (13, "clubs")   => "King of clubs",
    ...
};
```

Благодаря применению *шаблонов* (см. раздел “Шаблоны” в главе 4) становятся возможными многие другие варианты.

Операторы итераций

Язык C# позволяет многоократно выполнять последовательность операторов с помощью операторов `while`, `do-while`, `for` и `foreach`.

Циклы `while` и `do-while`

Циклы `while` многократно выполняют код в своем теле до тех пор, пока результатом выражения типа `bool` является `true`. Выражение проверяется *перед* выполнением тела цикла. Например, следующий код выводит 012:

```
int i = 0;
while (i < 3)
{
    Console.Write (i);
    i++;
}
```

Циклы `do-while` отличаются по функциональности от циклов `while` только тем, что выражение в них проверяется *после* выполнения блока операторов (гарантируя выполнение блока минимум один раз). Ниже приведен предыдущий пример, переписанный для применения цикла `do-while`:

```
int i = 0;
do
{
    Console.WriteLine (i);
    i++;
}
while (i < 3);
```

Циклы `for`

Циклы `for` похожи на циклы `while`, но имеют специальные конструкции для инициализации и итерации переменной цикла. Цикл `for` содержит три конструкции:

```
for (конструкция-инициализации; конструкция-условия; конструкция-итерации)
    оператор-или-блок-операторов
```

Все конструкции описаны ниже.

- **Конструкция инициализации.** Выполняется перед началом цикла; служит для инициализации одной или большего количества переменных *итерации*.
- **Конструкция условия.** Выражение типа `bool`, при значении `true` которого будет выполняться тело цикла.
- **Конструкция итерации.** Выполняется *после* каждого прохода блока операторов; обычно используется для обновления переменной итерации.

Например, следующий цикл выводит числа от 0 до 2:

```
for (int i = 0; i < 3; i++)
    Console.WriteLine (i);
```

Приведенный ниже код выводит первые 10 чисел последовательности Фибоначчи (в которой каждое число является суммой двух предыдущих):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
    Console.WriteLine (prevFib);
    int newFib = prevFib + curFib;
    prevFib = curFib; curFib = newFib;
}
```

Любую из трех частей оператора `for` разрешено опускать. Вот как можно было бы реализовать бесконечный цикл (хотя подойдет и `while(true)`):

```
for (;;)
    Console.WriteLine ("interrupt me");
```

Циклы `foreach`

Оператор `foreach` обеспечивает проход по всем элементам в перечислимом объекте. Большинство типов .NET, которые представляют набор или список элементов, являются перечислимыми. Примерами перечислимых типов могут служить массивы и строки. Ниже приведен код для перечисления символов в строке, от первого до последнего:

```
foreach (char c in "beer") // c - переменная итерации
    Console.WriteLine (c);
```

Вот как выглядит вывод:

```
b
e
e
r
```

Перечислимые объекты описаны в разделе “Перечисление и итераторы” главы 4.

Операторы перехода

К операторам перехода в C# относятся `break`, `continue`, `goto`, `return` и `throw`.



Операторы перехода подчиняются правилам надежности операторов `try` (см. раздел “Операторы `try` и исключения” в главе 4). Это означает следующее:

- при переходе из блока `try` перед достижением цели перехода всегда выполняется блок `finally` оператора `try`;
- переход не может производиться изнутри блока `finally` наружу (кроме как через `throw`).

Оператор `break`

Оператор `break` заканчивает выполнение тела итерации или оператора `switch`:

```
int x = 0;
while (true)
{
    if (x++ > 5)
        break; // Прервать цикл
}
// После break выполнение продолжится здесь
...
```

Оператор `continue`

Оператор `continue` игнорирует оставшиеся операторы в цикле и начинает следующую итерацию. В представленном ниже цикле пропускаются четные числа:

```
for (int i = 0; i < 10; i++)
{
    if ((i % 2) == 0)      // Если значение i четное,
        continue;          // тогда перейти к следующей итерации
    Console.Write (i + " ");
}
```

Вот вывод:

1 3 5 7 9

Оператор `goto`

Оператор `goto` переносит выполнение на указанную метку внутри блока операторов. Он имеет следующую форму:

```
goto метка-оператора;
```

или же такую форму, когда используется внутри оператора `switch`:

```
goto case константа-case; // (Работает только с константами, но не с шаблонами)
```

Метка — это заполнитель в блоке кода, который предваряет оператор и завершается двоеточием.

Показанный далее код выполняет итерацию по числам от 1 до 5, имитируя поведение цикла `for`:

```
int i = 1;  
startLoop:  
if (i <= 5)  
{  
    Console.Write (i + " ");  
    i++;  
    goto startLoop;  
}
```

Ниже приведен вывод:

```
1 2 3 4 5
```

Форма `goto case` константа-`case` переносит выполнение на другую конструкцию `case` в блоке `switch` (см. раздел “Оператор `switch`” ранее в главе).

Оператор `return`

Оператор `return` завершает метод и должен возвращать выражение возвращаемого типа метода, если метод не является `void`:

```
decimal AsPercentage (decimal d)  
{  
    decimal p = d * 100m;  
    return p;           // Возвратиться в вызывающий метод со значением  
}
```

Оператор `return` может находиться в любом месте метода (кроме блока `finally`) и встречаться более одного раза.

Оператор `throw`

Оператор `throw` генерирует исключение для указания на то, что возникла ошибка (см. раздел “Операторы `try` и исключения” в главе 4):

```
if (w == null)  
    throw new ArgumentNullException (...);
```

Смешанные операторы

Оператор `using` предлагает элегантный синтаксис для вызова метода `Dispose` на объектах, которые реализуют интерфейс `IDisposable`, внутри блока `finally` (см. раздел “Операторы `try` и исключения” в главе 4 и раздел “`IDisposable`, `Dispose` и `Close`” в главе 12).



Ключевое слово `using` в C# перегружено, поэтому в разных контекстах оно имеет разный смысл. В частности, директива `using` отличается от оператора `using`.

Оператор `lock` является сокращением для вызова методов `Enter` и `Exit` класса `Monitor` (см. главы 14 и 23).

Пространства имен

Пространство имен — это область, предназначенная для имен типов. Типы обычно организуются в иерархические пространства имен, облегчая их поиск и устраняя возможность возникновения конфликтов. Например, тип RSA, который поддерживает шифрование открытым ключом, определен в следующем пространстве имен:

```
System.Security.Cryptography
```

Пространство имен — неотъемлемая часть имени типа. В приведенном ниже коде вызывается метод Create класса RSA:

```
System.Security.Cryptography.RSA rsa =  
    System.Security.Cryptography.RSA.Create();
```



Пространства имен не зависят от сборок, которые являются файлами .dll, служащими единицами развертывания (см. главу 17).

Пространства имен также не влияют на видимость членов — public, internal, private и т.д.

Ключевое слово namespace определяет пространство имен для типов внутри данного блока. Например:

```
namespace Outer.Middle.Inner  
{  
    class Class1 {}  
    class Class2 {}  
}
```

С помощью точек отражается иерархия вложенных пространств имен. Следующий код семантически идентичен коду из предыдущего примера:

```
namespace Outer  
{  
    namespace Middle  
    {  
        namespace Inner  
        {  
            class Class1 {}  
            class Class2 {}  
        }  
    }  
}
```

Ссылаться на тип можно с помощью его *полностью заданного имени*, которое включает все пространства имен, от самого внешнего до самого внутреннего. Например, мы могли бы сослаться на Class1 из предшествующего примера в форме Outer.Middle.Inner.Class1.

Говорят, что типы, которые не определены в каком-либо пространстве имен, находятся в *глобальном пространстве имен*. Глобальное пространство имен также включает пространства имен верхнего уровня, такие как Outer в приведенном примере.

Пространства имен с областью видимости на уровне файлов

Часто необходимо, чтобы все типы в файле были определены в одном пространстве имен:

```
namespace MyNamespace
{
    class Class1 {}
    class Class2 {}
}
```

Начиная с версии C# 10, этого можно добиться с помощью *пространства имен с областью видимости на уровне файла*:

```
namespace MyNamespace; // Применяется ко всем последующим типам,
                      // определенным в файле
class Class1 {}        // Внутри MyNamespace
class Class2 {}        // Внутри MyNamespace
```

Пространства имен с областью видимости на уровне файла уменьшают беспорядок и устраниют ненужный уровень отступов.

Директива `using`

Директива `using` импортирует пространство имен, позволяя ссылаться на типы без указания их полностью заданных имен. В следующем коде импортируется пространство имен `Outer.Middle.Inner` из предыдущего примера:

```
using Outer.Middle.Inner;
Class1 c; // Полностью заданное имя указывать не обязательно
```



Вполне законно (и часто желательно) определять в двух разных пространствах имен одно и то же имя типа. Тем не менее, обычно это делается только в случае низкой вероятности возникновения ситуации, когда потребитель пожелает импортировать сразу оба пространства имен. Хорошим примером может служить класс `TextBox`, который определен и в `System.Windows.Controls` (WPF), и в `System.Windows.Forms` (Windows Forms).

Директиву `using` можно вкладывать внутрь самого пространства имен, чтобы ограничивать область ее действия.

Директива `global using`

Начиная с версии C# 10, в случае добавления к директиве `using` ключевого слова `global` директива будет применяться ко всем файлам в проекте или в единице компиляции:

```
global using System;
global using System.Collections.Generic;
```

Это позволяет централизовать общее импортирование и избежать повторения одних и тех же директив в каждом файле.

Директивы `global using` должны предшествовать неглобальным директивам и не могут находиться внутри объявлений пространств имен. Директиву `global using` можно использовать вместе с `using static`.

Неявные директивы `global using`

Начиная с версии .NET 6, файлы проектов позволяют применять неявные директивы `global using`. Если для элемента `ImplicitUsings` в файле проекта установлено значение `true` (по умолчанию так принято для новых проектов), то автоматически импортируются следующие пространства имен:

```
System
System.Collections.Generic
System.IO
System.Linq
System.Net.Http
System.Threading
System.Threading.Tasks
```

Дополнительные пространства имен импортируются на основе комплекта SDK проекта (Web, Windows Forms, WPF и т.д.).

Директива `using static`

Директива `using static` импортирует *тип*, а не пространство имен. Затем все статические члены импортированного типа можно использовать, не снабжая их именем типа. В показанном ниже примере вызывается статический метод `WriteLine` класса `Console` без ссылки на тип:

```
using static System.Console;
WriteLine ("Hello");
```

Директива `using static` импортирует все доступные статические члены типа, включая поля, свойства и вложенные типы (глава 3). Ее также можно применять к перечислимым типам (см. главу 3), что приведет к импортированию их членов. Таким образом, если мы импортируем следующий перечислимый тип:

```
using static System.Windows.Visibility;
```

то вместо `Visibility.Hidden` сможем указывать просто `Hidden`:

```
var textBox = new TextBox { Visibility = Hidden }; // Стиль XAML
```

Если между несколькими директивами `using static` возникнет неоднозначность, тогда компилятор C# не сумеет вывести корректный тип из контекста и сообщит об ошибке.

Правила внутри пространства имен

Область видимости имен

Имена, объявленные во внешних пространствах имен, могут использоваться во внутренних пространствах имен без дополнительного указания пространства.

В следующем примере `Class1` не нуждается в указании пространства имен внутри `Inner`:

```
namespace Outer
{
    class Class1 {}
    namespace Inner
    {
        class Class2 : Class1 {}
    }
}
```

Если ссылаться на тип необходимо из другой ветви иерархии пространств имен, тогда можно применять частично заданное имя. В показанном ниже примере класс SalesReport основан на Common.ReportBase:

```
namespace MyTradingCompany
{
    namespace Common
    {
        class ReportBase {}
    }
    namespace ManagementReporting
    {
        class SalesReport : Common.ReportBase {}
    }
}
```

Сокрытие имен

Если одно и то же имя типа встречается и во внутреннем, и во внешнем пространстве имен, то преимущество получает тип из внутреннего пространства имен. Чтобы сослаться на тип из внешнего пространства имен, имя потребуется уточнить. Например:

```
namespace Outer
{
    class Foo { }

    namespace Inner
    {
        class Foo { }
        class Test
        {
            Foo f1;           // = Outer.Inner.Foo
            Outer.Foo f2;     // = Outer.Foo
        }
    }
}
```



Все имена типов на этапе компиляции преобразуются в полностью заданные имена. Код на промежуточном языке (IL) не содержит неполные или частично заданные имена.

Повторяющиеся пространства имен

Объявление пространства имен можно повторять при условии, что имена типов внутри объявлений не конфликтуют друг с другом:

```
namespace Outer.Middle.Inner
{
    class Class1 {}
}
namespace Outer.Middle.Inner
{
    class Class2 {}
}
```

Код в этом примере можно даже разнести по двум исходным файлам, что позволит компилировать каждый класс в отдельную сборку.

Исходный файл 1:

```
namespace Outer.Middle.Inner
{
    class Class1 {}
}
```

Исходный файл 2:

```
namespace Outer.Middle.Inner
{
    class Class2 {}
}
```

Вложенные директивы `using`

Директиву `using` можно вкладывать внутрь пространства имен, что позволяет ограничивать область видимости директивы `using` объявлением пространства имен. В следующем примере имя `Class1` доступно в одной области видимости, но не доступно в другой:

```
namespace N1
{
    class Class1 {}
}
namespace N2
{
    using N1;
    class Class2 : Class1 {}
}
namespace N2
{
    class Class3 : Class1 {} // Ошибка на этапе компиляции
}
```

Назначение псевдонимов типам и пространствам имен

Импортирование пространства имен может привести к конфликту имен типов. Вместо полного пространства имен можно импортировать только конкретные типы, которые нужны, и назначать каждому типу псевдоним:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

Псевдоним можно назначить целому пространству имен:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

Назначение псевдонима любому типу (C# 12)

Начиная с версии C# 12, посредством директивы `using` можно назначать псевдоним любому типу, включая, например, массивы:

```
using NumberList = double[];
NumberList numbers = { 2.5, 3.5 };
```

Можно также назначать псевдонимы кортежам, как будет показано в разделе “Назначение псевдонимов кортежам (C# 12)” главы 4.

Дополнительные возможности пространств имен

Внешние псевдонимы

Внешние псевдонимы позволяют программе ссылаться на два типа с одним и тем же полностью заданным именем (т.е. сочетания пространства имен и имени типа одинаковы). Этот необычный сценарий может возникнуть только в ситуации, когда два типа поступают из разных сборок. Рассмотрим представленный ниже пример.

Библиотека 1, скомпилированная в Widgets1.dll:

```
namespace Widgets
{
    public class Widget {}
}
```

Библиотека 2, скомпилированная в Widgets2.dll:

```
namespace Widgets
{
    public class Widget {}
}
```

Приложение, которое ссылается на Widgets1.dll и Widgets2.dll:

```
using Widgets;
Widget w = new Widget();
```

Код такого приложения не может быть скомпилирован, потому что имеется неоднозначность с `Widget`. Решить проблему неоднозначности в приложении помогут внешние псевдонимы.

Первым делом нужно изменить файл `.csproj`, назначив каждой ссылке уникальный псевдоним:

```
<ItemGroup>
<Reference Include="Widgets1">
    <Aliases>W1</Aliases>
</Reference>
<Reference Include="Widgets2">
    <Aliases>W2</Aliases>
</Reference>
</ItemGroup>
```

Затем необходимо воспользоваться директивой `extern alias`:

```
extern alias W1;
extern alias W2;
W1.Widgets.Widget w1 = new W1.Widgets.Widget();
W2.Widgets.Widget w2 = new W2.Widgets.Widget();
```

Уточнители псевдонимов пространств имен

Как упоминалось ранее, имена во внутренних пространствах имен скрывают имена из внешних пространств. Тем не менее, иногда даже применение полностью заданного имени не устраниет конфликт. Взгляните на следующий пример:

```
namespace N
{
    class A
    {
        static void Main() => new A.B();           // Создать экземпляр класса B
        public class B {}                         // Вложенный тип
    }
}
namespace A
{
    class B {}
}
```

Метод Main мог бы создавать экземпляр либо вложенного класса B, либо класса B из пространства имен A. Компилятор всегда назначает более высокий приоритет идентификаторам из текущего пространства имен (в данном случае — вложенному классу B).

Для разрешения конфликтов подобного рода название пространства имен может быть уточнено относительно одного из следующих аспектов:

- глобального пространства имен — корня всех пространств имен (идентифицируется контекстным ключевым словом `global`);
- набора внешних псевдонимов.

Псевдоним пространства имен уточняется с помощью маркера `::`. В этом примере мы уточняем с использованием глобального пространства имен (чаще всего такое уточнение можно встречать в автоматически генерируемом коде — оно направлено на устранение конфликтов имен):

```
namespace N
{
    class A
    {
        static void Main()
        {
            System.Console.WriteLine (new A.B());
            System.Console.WriteLine (new global::A.B());
        }
        public class B {}
    }
}
namespace A
{
    class B {}
}
```

Ниже приведен пример уточнения псевдонима (взятый из примера в разделе “Внешние псевдонимы” ранее в главе):

```
extern alias W1;
extern alias W2;
W1::Widgets.Widget w1 = new W1::Widgets.Widget();
W2::Widgets.Widget w2 = new W2::Widgets.Widget();
```



Создание типов в языке C#

В настоящей главе мы займемся исследованием типов и членов типов.

Классы

Класс является наиболее распространенной разновидностью ссылочного типа. Простейшее из возможных объявление класса выглядит следующим образом:

```
class YourClassName
{
}
```

Более сложный класс может дополнительно иметь перечисленные ниже компоненты.

Перед ключевым словом `class`

Атрибуты и модификаторы класса. К модификаторам невложенных классов относятся `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe` и `partial`

После `YourClassName`

Параметры и ограничения обобщенных типов, базовый класс и интерфейсы

Внутри фигурных скобок

Члены класса (к ним относятся методы, свойства, индексаторы, события, поля, конструкторы, перегруженные операции, вложенные типы и финализатор)

В текущей главе описаны все конструкции кроме атрибутов, функций операций и ключевого слова `unsafe`, которые раскрываются в главе 4. В последующих разделах все члены класса рассматриваются по очереди.

Поле — это переменная, которая является членом класса или структуры; например:

```
class Octopus
{
    string name;
    public int Age = 10;
}
```

С полями разрешено применять следующие модификаторы.

Статический модификатор	static
Модификаторы доступа	public internal private protected
Модификатор наследования	new
Модификатор небезопасного кода	unsafe
Модификатор доступа только для чтения	readonly
Модификатор многопоточности	volatile

Для закрытых полей существуют два популярных соглашения об именовании: “верблюжий” стиль (например, `firstName`) и “верблюжий” стиль с подчеркиванием (`_firstName`). Последнее соглашение позволяет мгновенно отличать закрытые поля от параметров и локальных переменных.

Модификатор `readonly`

Модификатор `readonly` предотвращает изменение поля после его создания. Присваивать значение полю, допускающему только чтение, можно только в его объявлении или внутри конструктора типа, где оно определено.

Инициализация полей

Инициализация полей необязательна. Неинициализированное поле получает свое стандартное значение (0, '\0', null, false). Инициализаторы полей выполняются перед конструкторами:

```
public int Age = 10;
```

Инициализатор поля может содержать выражения и вызывать методы:

```
static readonly string TempFolder = System.IO.Path.GetTempPath();
```

Объявление множества полей вместе

Для удобства множество полей одного типа можно объявлять в списке, разделяя их запятыми. Это подходящий способ обеспечить совместное использование всеми полями одних и тех же атрибутов и модификаторов полей:

```
static readonly int legs = 8,  
eyes = 2;
```

Константа вычисляется статически на этапе компиляции и компилятор литеральным образом подставляет ее значение всякий раз, когда она встречается (что довольно похоже на макрос в C++). Константа может относиться к любому из встроенных числовых типов, `bool`, `char`, `string` или к типу перечисления.

Константа объявляется с помощью ключевого слова `const` и должна быть инициализирована каким-нибудь значением. Например:

```
public class Test  
{  
    public const string Message = "Hello World";  
}
```

Константа может выполнять роль, сходную с ролью поля `static readonly`, но она гораздо более ограничена — как в типах, которые можно применять,

так и в семантике инициализации поля. Константа также отличается от поля static readonly тем, что ее вычисление происходит на этапе компиляции. Соответственно показанный ниже код:

```
public static double Circumference (double radius)
{
    return 2 * System.Math.PI * radius;
}
```

компилируется в такой код:

```
public static double Circumference (double radius)
{
    return 6.2831853071795862 * radius;
}
```

Имеет смысл, чтобы поле PI было константой, т.к. его значение никогда не меняется. Напротив, поле static readonly может получать отличающееся значение при каждом запуске программы:

```
static readonly DateTime StartupTime = DateTime.Now;
```



Поле static readonly также полезно, когда другим сборкам открывается доступ к значению, которое в более поздней версии сборки может измениться. Например, пусть сборка X открывает доступ к константе:

```
public const decimal ProgramVersion = 2.3;
```

Если сборка Y ссылается на сборку X и пользуется константой ProgramVersion, тогда при компиляции значение 2.3 будет встроено в сборку Y. В результате, когда сборка X позже перекомпилируется с константой ProgramVersion, установленной в 2.4, в сборке Y по-прежнему будет применяться старое значение 2.3 *до тех пор, пока сборка Y не будет перекомпилирована*. Поле static readonly позволяет избежать такой проблемы.

На описанную ситуацию можно взглянуть и по-другому: любое значение, которое способно измениться в будущем, по определению не является константой, а потому не должно быть представлено как константа.

Константы можно также объявлять локально внутри метода:

```
void Test()
{
    const double twoPI = 2 * System.Math.PI;
    ...
}
```

Нелокальные константы допускают использование перечисленных далее модификаторов.

Модификаторы доступа

public internal private protected

Модификатор наследования

new

Методы

Метод выполняет действие, представленное в виде последовательности операторов. Метод может получать *входные* данные из вызывающего кода посредством указания *параметров* и возвращать *выходные* данные обратно вызывающему коду за счет указания *возвращаемого типа*. Для метода может быть определен возвращаемый тип `void`, который говорит о том, что метод никакого значения не возвращает. Метод также может возвращать выходные данные вызывающему коду через параметры `ref/out`.

Сигнатура метода должна быть уникальной в рамках типа. Сигнатура метода включает в себя имя метода и типы параметров по порядку (но не имена параметров и не возвращаемый тип).

С методами разрешено применять следующие модификаторы.

Статический модификатор	<code>static</code>
Модификаторы доступа	<code>public internal private protected</code>
Модификаторы наследования	<code>new virtual abstract override sealed</code>
Модификатор частичного метода	<code>partial</code>
Модификаторы неуправляемого кода	<code>unsafe extern</code>
Модификатор асинхронного кода	<code>async</code>

Методы, сжатые до выражений

Метод, который состоит из единственного выражения, такой как:

```
int Foo (int x) { return x * 2; }
```

можно записать более кратко в виде *метода, сжатого до выражения* (*expression-bodied*). Фигурные скобки и ключевое слово `return` заменяются комбинацией `=>`:

```
int Foo (int x) => x * 2;
```

Функции, сжатые до выражений, могут также иметь возвращаемый тип `void`:

```
void Foo (int x) => Console.WriteLine (x);
```

Локальные методы

Метод можно определять внутри другого метода:

```
void WriteCubes ()  
{  
    Console.WriteLine (Cube (3));  
    Console.WriteLine (Cube (4));  
    Console.WriteLine (Cube (5));  
  
    int Cube (int value) => value * value * value;  
}
```

Локальный метод (`Cube` в данном случае) будет видимым только для охватывающего метода (`WriteCubes`). Это упрощает содержащий тип и немедленно подает сигнал любому, кто просматривает код, что `Cube` больше нигде не при-

меняется. Еще одно преимущество локальных методов заключается в том, что они могут обращаться к локальным переменным и параметрам охватывающего метода. С такой особенностью связано несколько последствий, которые описаны в разделе “Захватывание внешних переменных” главы 4.

Локальные методы могут появляться внутри функций других видов, таких как средства доступа к свойствам, конструкторы и т.д. Локальные методы можно даже помещать внутрь других локальных методов и лямбда-выражений, которые используют блок операторов (см. главу 4). Локальные методы могут быть итераторными (см. главу 4) или асинхронными (см. главу 14).

Статические локальные методы

Добавление модификатора `static` к локальному методу (возможность, появившаяся в версии C# 8) запрещает ему видеть локальные переменные и параметры охватывающего метода. Это помогает ослабить связность и предотвращает случайную ссылку в локальном методе на переменные из содержащего метода.

Локальные методы и операторы верхнего уровня

Любые методы, которые объявляются в операторах верхнего уровня, трактуются как локальные методы. Таким образом, они могут обращаться к переменным в операторах верхнего уровня (если только не были помечены как `static`):

```
int x = 3;  
Foo();  
void Foo() => Console.WriteLine (x);
```

Перегрузка методов



Локальные методы перегружать нельзя. Это значит, что методы, объявленные в операторах верхнего уровня (и трактуемые как локальные), не могут быть перегружены.

Тип может *перегружать* методы (определять несколько методов с одним и тем же именем) при условии, что их сигнатуры будут отличаться. Например, все перечисленные ниже методы могут сосуществовать внутри того же самого типа:

```
void Foo (int x) {...}  
void Foo (double x) {...}  
void Foo (int x, float y) {...}  
void Foo (float x, int y) {...}
```

Тем не менее, следующие пары методов не могут сосуществовать в рамках одного типа, поскольку возвращаемый тип и модификатор `params` не входят в состав сигнатур метода:

```
void Foo (int x) {...}  
float Foo (int x) {...} // Ошибка на этапе компиляции  
void Goo (int[] x) {...}  
void Goo (params int[] x) {...} // Ошибка на этапе компиляции
```

Способ передачи параметра — по значению или по ссылке — также является частью сигнатуры. Скажем, Foo(int) может сосуществовать вместе с Foo(ref int) или Foo(out int). Однако Foo(ref int) и Foo(out int) существовать не могут:

```
void Foo (int x) {...}
void Foo (ref int x) {...}           // До этого места все в порядке
void Foo (out int x) {...}          // Ошибка на этапе компиляции
```

Конструкторы экземпляров

Конструкторы выполняют код инициализации класса или структуры. Конструктор определяется подобно методу за исключением того, что вместо имени метода и возвращаемого типа указывается имя типа, к которому относится этот конструктор:

```
Panda p = new Panda ("Petey");      // Вызов конструктора
public class Panda
{
    string name;                  // Определение поля
    public Panda (string n)       // Определение конструктора
    {
        name = n;                // Код инициализации (установка поля)
    }
}
```

Конструкторы экземпляров допускают применение следующих модификаторов.

Модификаторы доступа	public internal private protected
Модификаторы неуправляемого кода	unsafe extern

Конструкторы, содержащие единственный оператор, также можно записывать как члены, сжатые до выражений:

```
public Panda (string n) => name = n;
```



Если имя параметра (или любое имя переменной) конфликтует с именем поля, то неоднозначность можно устраниить, предварив имя поля ссылкой **this**:

```
public Panda (string name) => this.name = name;
```

Перегрузка конструкторов

Класс или структура может перегружать конструкторы. Один перегруженный конструктор способен вызывать другой, используя ключевое слово **this**:

```
public class Wine
{
    public decimal Price;
    public int Year;
    public Wine (decimal price) => Price = price;
    public Wine (decimal price, int year) : this (price) => Year = year;
}
```

Когда один конструктор вызывает другой, то первым выполняется **вызванный конструктор**. Другому конструктору можно передавать **выражение**:

```
public Wine (decimal price, DateTime year) : this (price, year.Year) { }
```

Выражение может обращаться к статическим членам класса, но не к членам экземпляра. (Причина в том, что на данной стадии объект еще не инициализирован конструктором, поэтому вызов любого метода вполне вероятно приведет к сбою.)



В этом конкретном примере более эффективной была бы реализация с одним конструктором, у которого год (*year*) является необязательным параметром:

```
public Wine (decimal price, int year = 0)
{
    Price = price; Year = year;
}
```

В разделе “Инициализаторы объектов” далее в главе будет предложено еще одно решение.

Неявные конструкторы без параметров

Компилятор C# автоматически генерирует для класса открытый конструктор без параметров, если и только если в нем не определено ни одного конструктора. Однако после определения хотя бы одного конструктора конструктор без параметров больше автоматически не генерируется.

Порядок выполнения конструктора и инициализации полей

Как было показано ранее, поля могут инициализироваться стандартными значениями в их объявлении:

```
class Player
{
    int shields = 50;           // Инициализируется первым
    int health = 100;          // Инициализируется вторым
}
```

Инициализация полей происходит *перед* выполнением конструктора в порядке их объявления.

Неоткрытые конструкторы

Конструкторы не обязательно должны быть открытыми. Распространенной причиной наличия неоткрытого конструктора является управление созданием экземпляров через вызов статического метода. Статический метод может использоваться для возвращения объекта из пула вместо создания нового объекта или для возвращения экземпляров разных подклассов на основе входных аргументов:

```
public class Class1
{
    Class1() {}           // Закрытый конструктор
    public static Class1 Create (...)

    {
        // Специальная логика для возвращения экземпляра Class1
        ...
    }
}
```

Деконструкторы

Деконструктор (также называемый *деконструирующим методом*) действует почти как противоположность конструктору: в то время когда конструктор обычно принимает набор значений (в качестве параметров) и присваивает их полям, деконструктор делает обратное, присваивая поля набору переменных.

Метод деконструирования должен называться `Deconstruct` и иметь один или большее количество параметров `out`, как в следующем классе:

```
class Rectangle
{
    public readonly float Width, Height;
    public Rectangle (float width, float height)
    {
        Width = width;
        Height = height;
    }
    public void Deconstruct (out float width, out float height)
    {
        width = Width;
        height = Height;
    }
}
```

Для вызова деконструктора применяется специальный синтаксис:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;           // Деконструирование
Console.WriteLine (width + " " + height);      // 3 4
```

Деконструирующий вызов содержится во второй строке. Он создает две локальные переменные и затем обращается к методу `Deconstruct`. Вот чему эквивалентен этот деконструирующий вызов:

```
float width, height;
rect.Deconstruct (out width, out height);
```

Или:

```
rect.Deconstruct (out var width, out var height);
```

Деконструирующие вызовы допускают неявную типизацию, так что наш вызов можно было бы сократить следующим образом:

```
(var width, var height) = rect;
```

Или просто так:

```
var (width, height) = rect;
```



Вы можете применить символ отбрасывания C# (`_`), если не заинтересованы в одной или большем количестве переменных:

```
var (_, height) = rect;
```

Это точнее отражает намерение, чем объявление переменной, которую вы никогда не будете использовать.

Если переменные, в которые производится деконструирование, уже определены, то типы вообще не указываются:

```
float width, height;  
(width, height) = rect;
```

Такое действие называется *деконструиющим присваиванием*. Вы можете применить деконструиующее присваивание для упрощения конструктора вашего класса:

```
public Rectangle (float width, float height) =>  
(Width, Height) = (width, height);
```

За счет перегрузки метода `Deconstruct` вы можете предложить вызывающему коду целый диапазон вариантов деконструирования.



Метод `Deconstruct` может быть расширяющим методом (см. раздел “Расширяющие методы” в главе 4). Это удобный прием, если вы хотите деконструировать типы, автором которых не являетесь.

Начиная с версии C# 10, при деконструировании можно смешивать и сопоставлять существующие и новые переменные:

```
double x1 = 0;  
(x1, double y2) = rect;
```

Инициализаторы объектов

Для упрощения инициализации объекта любые его доступные поля и свойства можно устанавливать с помощью *инициализатора объекта* прямо после создания. Например, рассмотрим следующий класс:

```
public class Bunny  
{  
    public string Name;  
    public bool LikesCarrots, LikesHumans;  
  
    public Bunny () {}  
    public Bunny (string n) => Name = n;  
}
```

Вот как можно создавать объекты `Bunny` с использованием инициализаторов объектов:

```
// Обратите внимание, что для конструкторов без параметров круглые  
// скобки можно не указывать  
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false } ;  
Bunny b2 = new Bunny ("Bo") { LikesCarrots=true, LikesHumans=false } ;
```

Код, конструирующий объекты `b1` и `b2`, в точности эквивалентен показанному ниже коду:

```
Bunny templ = new Bunny(); // templ - имя, сгенерированное компилятором  
templ.Name = "Bo";  
templ.LikesCarrots = true;  
templ.LikesHumans = false;  
Bunny b1 = templ;
```

```
Bunny temp2 = new Bunny ("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

Временные переменные предназначены для гарантии того, что если в течение инициализации произойдет исключение, то вы в итоге не получите наполовину инициализированный объект.

Сравнение инициализаторов объектов и необязательных параметров

Вместо того чтобы полагаться на инициализаторы объектов, конструктор класса `Bunny` можно было бы реализовать с одним обязательным и двумя необязательными параметрами, как показано ниже:

```
public Bunny (string name,
              bool likesCarrots = false,
              bool likesHumans = false)
{
    Name = name;
    LikesCarrots = likesCarrots;
    LikesHumans = likesHumans;
}
```

Тогда у нас появилась бы возможность конструировать экземпляр `Bunny` следующим образом:

```
Bunny b1 = new Bunny (name: "Bo",
                      likesCarrots: true);
```

Исторически применение конструкторов для инициализации объектов мог обеспечивать преимущество, которое заключалось в том, что при желании поля класса `Bunny` (или *свойства*, как вскоре будет объяснено) можно было бы сделать доступными только для чтения. Превращать поля или свойства в допускающие только чтение рекомендуется тогда, когда нет законного основания для их изменения в течение времени жизни объекта. Тем не менее, как будет показано далее при обсуждении свойств, модификатор `init`, появившийся в версии C# 9, позволяет достичь той же цели с помощью инициализаторов объектов.

Необязательные параметры обладают двумя недостатками. Первый недостаток связан с тем, что хотя использование необязательных параметров делает возможными типы только для чтения, легко реализовать *неразрушающее изменение* с их помощью не удастся. (Неразрушающее изменение и решение этой проблемы будут раскрыты в разделе “Записи” главы 4.)

Второй недостаток необязательных параметров заключается в том, что в случае применения в открытых библиотеках они препятствуют обратной совместимости. Причина в том, что добавление необязательного параметра в более позднее время нарушает *двоичную совместимость* сборки с существующими потребителями. (Это особенно важно при опубликовании библиотеки как пакета NuGet: проблема становится трудной для решения, когда потребитель ссылается на пакеты *A* и *B*, а каждый из них зависит от несовместимых версий пакета *L*.)

Трудность в том, что значение каждого необязательного параметра внедряется в *место вызова*.

Другими словами, компилятор C# транслирует показанный выше вызов конструктора в такой код:

```
Bunny b1 = new Bunny ("Bo", true, false);
```

Проблема возникает, когда мы создаем экземпляр класса Bunny из другой сборки и позже модифицируем этот класс, добавив еще один необязательный параметр (скажем, likesCats). Если не перекомпилироватьзывающуюся сборку, то она продолжит вызывать (уже несуществующий) конструктор с тремя параметрами, приводя к ошибке во время выполнения. (Более тонкая проблема связана с тем, что даже когда мы изменяем значение одного из необязательных параметров,зывающий код в других сборках продолжит пользоваться старым необязательным значением до тех пор, пока эти сборки не будут перекомпилированы.)

И последнее соображение касается влияния конструкторов на создание подклассов (которое будет обсуждаться в разделе “Наследование” далее в главе). Наличие нескольких конструкторов с длинными списками параметров делает создание подклассов громоздким; следовательно, это может помочь свести к минимуму количество и сложность конструкторов и применять инициализаторы объектов для заполнения деталей.

Ссылка `this`

Ссылка `this` указывает на сам экземпляр. В следующем примере метод Marry применяет ссылку `this` для установки поля Mate экземпляра partner:

```
public class Panda
{
    public Panda Mate;
    public void Marry (Panda partner)
    {
        Mate = partner;
        partner.Mate = this;
    }
}
```

Ссылка `this` также устраняет неоднозначность между локальной переменной или параметром и полем, например:

```
public class Test
{
    string name;
    public Test (string name) { this.name = name; }
}
```

Использование ссылки `this` допускается только внутри нестатических членов класса или структуры.

Свойства

Снаружи свойства выглядят похожими на поля, но подобно методам внутренне они содержат логику. Скажем, взглянув на следующий код, невозможно сказать, чем является `CurrentPrice` — полем или свойством:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Свойство объявляется как поле, но с добавлением блока `get/set`. Реализовать `CurrentPrice` в виде свойства можно так:

```
public class Stock
{
    decimal currentPrice;           // Закрытое "поддерживающее" поле
    public decimal CurrentPrice     // Открытое свойство
    {
        get { return currentPrice; }
        set { currentPrice = value; }
    }
}
```

С помощью `get` и `set` обозначаются *средства доступа* к свойству. Средство доступа `get` запускается при чтении свойства. Оно должно возвращать значение, имеющее тип как у самого свойства. Средство доступа `set` выполняется во время присваивания свойству значения. Оно принимает неявный параметр по имени `value` с типом свойства, который обычно присваивается какому-то закрытому полю (в данном случае `currentPrice`).

Хотя доступ к свойствам осуществляется таким же способом, как к полям, свойства отличаются тем, что предоставляют программисту полный контроль над получением и установкой их значений. Такой контроль позволяет программисту выбрать любое необходимое внутреннее представление, не раскрывая внутренние детали потребителю свойства. В приведенном примере метод `set` мог бы генерировать исключение, если значение `value` выходит за пределы допустимого диапазона.



В настоящей книге повсеместно применяются открытые поля, чтобы излишне не усложнять примеры и не отвлекать от их сути. В реальном приложении для содействия инкапсуляции предпочтение обычно отдается открытым свойствам, а не открытым полям.

Со свойствами разрешено применять следующие модификаторы.

Статический модификатор	<code>static</code>
Модификаторы доступа	<code>public internal private protected</code>
Модификаторы наследования	<code>new virtual abstract override sealed</code>
Модификаторы неуправляемого кода	<code>unsafe extern</code>

Свойства только для чтения и вычисляемые свойства

Свойство разрешает только чтение, если для него указано лишь средство доступа `get`, и только запись, если определено лишь средство доступа `set`. Свойства только для записи используются редко.

Свойство обычно имеет отдельное поддерживающее поле, предназначенное для хранения внутренних данных. Тем не менее, свойство может также возвращать значение, вычисленное на основе других данных:

```
decimal currentPrice, sharesOwned;  
public decimal Worth  
{  
    get { return currentPrice * sharesOwned; }  
}
```

Свойства, сжатые до выражений

Свойство только для чтения вроде показанного в предыдущем примере можно объявлять более кратко в виде *свойства, сжатого до выражения*. Фигурные скобки и ключевые слова `get` и `return` заменяются комбинацией `=>`:

```
public decimal Worth => currentPrice * sharesOwned;
```

С помощью небольшого дополнительного синтаксиса средства доступа `set` также можно объявлять как сжатые до выражения:

```
public decimal Worth  
{  
    get => currentPrice * sharesOwned;  
    set => sharesOwned = value / currentPrice;  
}
```

Автоматические свойства

Наиболее распространенная реализация свойства предусматривает наличие средств доступа `get` и/или `set`, которые просто читают и записывают в закрытое поле того же типа, что и свойство. Объявление *автоматического свойства* указывает компилятору на необходимость предоставления такой реализации. Первый пример в этом разделе можно усовершенствовать, объявив `CurrentPrice` как автоматическое свойство:

```
public class Stock  
{  
    ...  
    public decimal CurrentPrice { get; set; }  
}
```

Компилятор автоматически создает закрытое поддерживающее поле со специальным генерированным именем, ссылаясь на которое невозможно. Средство доступа `set` может быть помечено как `private` или `protected`, если свойство должно быть доступно только для чтения другим типам. Автоматические свойства появились в версии C# 3.0.

Инициализаторы свойств

Как и поля, автоматические свойства можно снабжать инициализаторами свойств:

```
public decimal CurrentPrice { get; set; } = 123;
```

В результате свойство `CurrentPrice` получает начальное значение 123. Свойства с инициализаторами могут допускать только чтение:

```
public int Maximum { get; } = 999;
```

Подобно полям, предназначенному только для чтения, автоматические свойства, допускающие только чтение, могут устанавливаться также в конструкторе типа, что удобно при создании *неизменяемых* (только для чтения) типов.

Доступность `get` и `set`

Средства доступа `get` и `set` могут иметь разные уровни доступа. В типичном сценарии применения есть свойство `public` с модификатором доступа `internal` или `private`, указанным для средства доступа `set`:

```
public class Foo
{
    private decimal x;
    public decimal X
    {
        get { return x; }
        private set { x = Math.Round (value, 2); }
    }
}
```

Обратите внимание, что само свойство объявлено с более либеральным уровнем доступа (`public` в данном случае), а к средству доступа, которое должно быть *менее доступным*, добавлен соответствующий модификатор.

Средства доступа только для инициализации

Начиная с версии C# 9, средство доступа к свойству можно объявлять как `init`, а не `set`:

```
public class Note
{
    public int Pitch { get; init; } = 20; // Средство доступа только
                                         // для инициализации
    public int Duration { get; init; } = 100; // Средство доступа только
                                              // для инициализации
}
```

Такие средства доступа *только для инициализации* действуют как свойства только для чтения за исключением того, что их можно также устанавливать через инициализатор объекта:

```
var note = new Note { Pitch = 50 };
```

После этого свойство не может быть изменено:

```
note.Pitch = 200; // Ошибка - средство доступа только для инициализации!
```

Свойство, допускающее только инициализацию, нельзя устанавливать даже изнутри класса, в котором оно определено, кроме как через его инициализатор свойства, конструктор или еще одно средство доступа только для инициализации.

В качестве альтернативы свойствам, допускающим только инициализацию, будет наличие свойств только для чтения, которые заполняются через конструктор:

```
public class Note
{
    public int Pitch { get; }
    public int Duration { get; }

    public Note (int pitch = 20, int duration = 100)
    {
        Pitch = pitch; Duration = duration;
    }
}
```

Когда класс должен быть частью открытой библиотеки, такой подход затрудняет ведение версий, поскольку добавление к конструктору необязательного параметра в более позднее время нарушит двоичную совместимость с потребителями (тогда как добавление нового свойства только для инициализации ничего не нарушает).



Свойства, допускающие только инициализацию, обладают еще одним существенным преимуществом, которое заключается в том, что они делают возможным неразрушающее изменение при использовании в сочетании с записями (см. раздел “Записи” в главе 4).

Как и обычные средства доступа `set`, средства доступа только для инициализации могут также предоставлять реализацию:

```
public class Note
{
    readonly int _pitch;
    public int Pitch { get => _pitch; init => _pitch = value; }
    ...
}
```

Обратите внимание, что поле `_pitch` предназначено только для чтения: средства доступа только для инициализации разрешают модифицировать поля `readonly` в собственных классах. (Без такой возможности поле `_pitch` должно было бы быть переменной, а класс не мог бы стать внутренне неизменяемым.)



Изменение средства доступа свойства с `init` на `set` (или наоборот) является критическим двоичным изменением: любому, кто ссылается на вашу сборку, придется перекомпилировать свою сборку.

Это не должно стать проблемой при создании полностью неизменяемых типов, т.к. вашему типу никогда не потребуются свойства с (записываемым) средством доступа `set`.

Реализация свойств в CLR

Средства доступа к свойствам C# внутренне компилируются в методы с именами `get_XXX` и `set_XXX`:

```
public decimal get_CurrentPrice {...}
public void set_CurrentPrice (decimal value) {...}
```

Средство доступа `init` обрабатывается подобно `set`, но с дополнительным флагом, который закодирован в метаданных “обязательного модификатора”

(modreq), связанных со средством доступа `set` (см. раздел “Свойства, допускающие только инициализацию” в главе 18).

Простые невиртуальные средства доступа к свойствам *встраиваются* компилятором JIT, устранивая любую разницу в производительности между доступом к свойству и доступом к полю. Встраивание представляет собой разновидность оптимизации, при которой вызов метода заменяется телом этого метода.

Индексаторы

Индексаторы обеспечивают естественный синтаксис для доступа к элементам в классе или структуре, которая инкапсулирует список либо словарь значений. Индексаторы похожи на свойства, но предусматривают доступ через аргумент индекса, а не через имя свойства. Класс `string` имеет индексатор, который позволяет получать доступ к каждому его значению `char` посредством индекса `int`:

```
string s = "hello";
Console.WriteLine (s[0]);      // 'h'
Console.WriteLine (s[3]);      // 'l'
```

Синтаксис использования индексаторов похож на синтаксис работы с массивами за исключением того, что аргумент (аргументы) индекса может быть любого типа (типов).

Индексаторы имеют те же модификаторы, что и свойства (см. раздел “Свойства” ранее в главе), и могут вызываться null-условным образом за счет помещения вопросительного знака перед открывающей квадратной скобкой (см. раздел “Операции для работы со значениями null” в главе 2):

```
string s = null;
Console.WriteLine (s?[0]);     // Ничего не выводится; ошибка не возникает
```

Реализация индексатора

Чтобы реализовать индексатор, понадобится определить свойство по имени `this`, указав аргументы в квадратных скобках:

```
class Sentence
{
    string[] words = "The quick brown fox".Split();
    public string this [int wordNum]      // индексатор
    {
        get { return words [wordNum]; }
        set { words [wordNum] = value; }
    }
}
```

Вот как можно было бы применять такой индексатор:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);           // fox
s[3] = "kangaroo";
Console.WriteLine (s[3]);           // kangaroo
```

В типе разрешено объявлять несколько индексаторов, каждый с параметрами разных типов. Индексатор также может принимать более одного параметра:

```
public string this [int arg1, string arg2]
{
    get { ... }
    set { ... }
}
```

Если опустить средство доступа `set`, то индексатор станет предназначенным только для чтения, к тому же его определение можно сократить с помощью синтаксиса, сжатого до выражения:

```
public string this [int wordNum] => words [wordNum];
```

Реализация индексаторов в CLR

Индексаторы внутренне компилируются в методы с именами `get_Item` и `set_Item`, как показано ниже:

```
public string get_Item (int wordNum) {...}
public void set_Item (int wordNum, string value) {...}
```

Использование индексов и диапазонов посредством индексаторов

Вы можете поддерживать индексы и диапазоны (см. раздел “Индексы и диапазоны” в главе 2) в собственных классах за счет определения индексатора с параметром типа `Index` или `Range`. Мы можем расширить предыдущий пример, добавив в класс `Sentence` следующие индексаторы:

```
public string this [Index index] => words [index];
public string[] this [Range range] => words [range];
```

В результате появится возможность записывать такой код:

```
Sentence s = new Sentence();
Console.WriteLine (s [^1]);           // fox
string[] firstTwoWords = s [..2];     // (The, quick)
```

Основные конструкторы (C# 12)

Начиная с версии C# 12, можно помещать список параметров непосредственно после объявления класса (или структуры):

```
class Person (string firstName, string lastName)
{
    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

Тем самым компилятору сообщается о том, что необходимо построить *основной конструктор* (primary constructor) с использованием *параметров основного конструктора* (`firstName` и `lastName`), поэтому создавать экземпляр класса можно следующим образом:

```
Person p = new Person ("Alice", "Jones");
p.Print();      // Alice Jones
```

Основные конструкторы полезны при прототипировании и в других простых сценариях. Альтернативой было бы определение полей и явное написание конструктора:

```
class Person // (без основных конструкторов)
{
    string firstName, lastName; // Объявления полей
    public Person (string firstName, string lastName) // Конструктор
    {
        this.firstName = firstName; // Присвоить значение полю
        this.lastName = lastName; // Присвоить значение полю
    }
    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

Конструктор, создаваемый компилятором C#, называется основным, поскольку любые дополнительные конструкторы, которые вы решите (явно) написать, должны вызывать его:

```
class Person (string firstName, string lastName)
{
    public Person (string firstName, string lastName, int age)
        : this (firstName, lastName) // Должен вызывать основной конструктор
    {
        ...
    }
}
```

Это гарантирует, что параметры основного конструктора *всегда заполняются*.



В C# также предоставляются записи, которые рассматриваются в разделе “Записи” главы 4. Записи тоже поддерживают основные конструкторы, но компилятор выполняет дополнительный шаг с записями и генерирует (по умолчанию) открытое свойство только для инициализации для каждого параметра основного конструктора. Если такое поведение желательно, тогда рассмотрите возможность использования записей вместо классов.

Основные конструкторы лучше всего подходят для простых сценариев из-за следующих ограничений:

- добавить дополнительный код инициализации в основной конструктор невозможно;
- хотя параметр основного конструктора легко представить как открытое свойство, встроить логику проверки достоверности не так легко, если только свойство не доступно только для чтения.

Основные конструкторы заменяют стандартный конструктор без параметров, который в противном случае сгенерировал бы компилятор C#.

Семантика основных конструкторов

Чтобы понять, как работают основные конструкторы, давайте выясним, каким образом ведет себя обычный конструктор:

```
class Person
{
    public Person (string firstName, string lastName)
    {
        ... делать что-нибудь с firstName и lastName
    }
}
```

Когда код внутри этого конструктора завершает выполнение, параметры `firstName` и `lastName` покидают область видимости и впоследствии становятся недоступными. Напротив, параметры основного конструктора не исчезают из области видимости и позже могут быть доступными в любом месте внутри класса на протяжении всего срока существования объекта.



Параметры основного конструктора — это специальные конструкции C#, а не поля, хотя в конечном итоге компилятор при необходимости генерирует скрытые поля для хранения их значений.

Основные конструкторы и инициализаторы полей/свойств

Доступность параметров основного конструктора распространяется на инициализаторы полей и свойств. В следующем примере инициализаторы полей и свойств применяются для присваивания `firstName` открытому полю, а `lastName` — открытому свойству:

```
class Person (string firstName, string lastName)
{
    public readonly string FirstName = firstName;           // Поле
    public string LastName { get; } = lastName;                 // Свойство
}
```

Маскирование параметров основного конструктора

Поля (или свойства) могут повторно использовать имена параметров основного конструктора:

```
class Person (string firstName, string lastName)
{
    readonly string firstName = firstName;
    readonly string lastName = lastName;
    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

В этом сценарии поле или свойство имеет приоритет, маскируя параметр основного конструктора, за исключением правой части инициализаторов полей и свойств (выделены полужирным).



Подобно обычным параметрам параметры основного конструктора допускают запись. Маскирование их одноименным полем только для чтения (как в приведенном примере) эффективно защищает их от последующей модификации.

Проверка достоверности параметров основного конструктора

Иногда полезно выполнять вычисления в инициализаторах полей:

```
new Person ("Alice", "Jones").Print(); // Alice Jones
class Person (string firstName, string lastName)
{
    public readonly string FullName = firstName + " " + lastName;
    public void Print() => Console.WriteLine (FullName);
}
```

В следующем примере версия lastName в верхнем регистре сохраняется в поле с тем же самым именем (маскируя исходное значение):

```
new Person ("Alice", "Jones").Print(); // Alice JONES
class Person (string firstName, string lastName)
{
    readonly string lastName = lastName.ToUpper();
    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

В разделе “Выражения throw” главы 4 будет описано, как генерировать исключения при возникновении таких сценариев, как недопустимые данные. Ниже иллюстрируется, каким образом это можно использовать с основными конструкторами для проверки lastName при создании, гарантируя тем самым, что оно не может быть null:

```
new Person ("Alice", null); // генерирует исключение ArgumentNullException
class Person (string firstName, string lastName)
{
    readonly string lastName = (lastName == null)
        ? throw new ArgumentNullException ("lastName")
        : lastName;
}
```

(Не забывайте, что код в инициализаторе поля или свойства выполняется при создании объекта, а не при доступе к полю или свойству.) В следующем примере параметр основного конструктора предоставляется в виде свойства для чтения и записи:

```
class Person (string firstName, string lastName)
{
    public string LastName { get; set; } = lastName;
}
```

Добавить проверку достоверности в этот пример не так-то просто, поскольку проверять приходится в двух местах: в методе доступа set свойства (реализованном вручную) и в инициализаторе свойства. (Та же проблема возникает, если свойство определено как предназначеннное только для инициализации.) На данном этапе проще отказаться от основных конструкторов и явно определять конструктор вместе с поддерживающими полями.

Статические конструкторы

Статический конструктор выполняется однократно для *типа*, а не однократно для *экземпляра*. В типе может быть определен только один статический конструктор, он не должен принимать параметры, а также обязан иметь то же имя, что и тип:

```
class Test
{
    static Test() { Console.WriteLine ("Type Initialized"); }
}
```

Исполняющая среда автоматически вызывает статический конструктор прямо перед тем, как тип начинает применяться. Вызов инициируется двумя действиями:

- создание экземпляра типа;
- доступ к статическому члену типа.

Для статических конструкторов разрешены только модификаторы `unsafe` и `extern`.



Если статический конструктор генерирует необработанное исключение (см. главу 4), то тип, к которому он относится, становится *непригодным* в жизненном цикле приложения.



Начиная с версии C# 9, можно определять также *инициализатор модуля*, который выполняется один раз для сборки (при ее первой загрузке). Чтобы определить инициализатор модуля, напишите статический метод `void` и примените к нему атрибут `[ModuleInitializer]`:

```
[System.Runtime.CompilerServices.ModuleInitializer]
internal static void InitAssembly()
{
    ...
}
```

Статические конструкторы и порядок инициализации полей

Инициализаторы статических полей запускаются непосредственно *перед* вызовом статического конструктора. Если тип не имеет статического конструктора, тогда инициализаторы полей будут выполнятся до того, как тип начнет использоваться — или *в любой момент раньше* по прихоти исполняющей среды.

Инициализаторы статических полей выполняются в порядке объявления полей, что демонстрируется в следующем примере, где поле X инициализируется значением 0, а поле Y — значением 3:

```
class Foo
{
    public static int X = Y;           // 0
    public static int Y = 3;           // 3
}
```

Если поменять местами эти два инициализатора полей, то оба поля будут инициализированы значением 3. В показанном ниже примере на экран выводится 0, а затем 3, потому что инициализатор поля, в котором создается экземпляр Foo, выполняется до того, как поле X инициализируется значением 3:

```
Console.WriteLine (Foo.X); // 3  
class Foo  
{  
    public static Foo Instance = new Foo();  
    public static int X = 3;  
    Foo() => Console.WriteLine (X); // 0  
}
```

Если поменять местами строки кода, выделенные полужирным, тогда на экран будет выводиться 3 и затем снова 3.

Статические классы

Класс, помеченный как `static`, не допускает создание экземпляров или подклассов и должен состоять исключительно из статических членов. Хорошими примерами статических классов могут служить `System.Console` и `System.Math`.

Финализаторы

Финализаторы представляют собой методы, предназначенные только для классов, которые выполняются до того, как сборщик мусора освободит память, занятую объектом с отсутствующими ссылками на него. Синтаксически финализатор записывается как имя класса, предваренное символом `~`:

```
class Class1  
{  
    ~Class1()  
    {  
        ...  
    }  
}
```

В действительности это синтаксис C# для переопределения метода `Finalize` класса `Object`, и компилятор развертывает его в следующее объявление метода:

```
protected override void Finalize()  
{  
    ...  
    base.Finalize();  
}
```

Сборка мусора и финализаторы подробно обсуждаются в главе 12.
Финализаторы допускают применение следующего модификатора.

Модификатор неуправляемого кода unsafe

Финализаторы, состоящие из единственного оператора, могут быть записаны с помощью синтаксиса сжатия до выражения:

```
~Class1() => Console.WriteLine ("Finalizing");
```

Частичные типы и методы

Частичные типы позволяют расщеплять определение типа обычно с разнесением его по нескольким файлам. Распространенный сценарий предполагает автоматическую генерацию частичного класса из какого-то другого источника (например, шаблона или визуального конструктора Visual Studio) и последующее его дополнение методами, написанными вручную:

```
// PaymentFormGen.cs - сгенерирован автоматически
partial class PaymentForm { ... }

// PaymentForm.cs - написан вручную
partial class PaymentForm { ... }
```

Каждый участник должен иметь объявление `partial`; показанный ниже код не является допустимым:

```
partial class PaymentForm {}
class PaymentForm {}
```

Участники не могут содержать конфликтующие члены. Скажем, конструктор с теми же самыми параметрами повторять нельзя. Частичные типы распознаются полностью компилятором, а это значит, что каждый участник типа должен быть доступным на этапе компиляции и располагаться в той же сборке.

Базовый класс может быть задан для единственного участника или для множества участников (при условии, что для каждого из них базовый класс будет тем же самым). Кроме того, для каждого участника можно независимо указывать интерфейсы, подлежащие реализации. Базовые классы и интерфейсы рассматриваются в разделах “Наследование” и “Интерфейсы” далее в главе.

Компилятор не гарантирует какого-то определенного порядка инициализации полей в рамках объявлений частичных типов.

Частичные методы

Частичный тип может содержать *частичные методы*. Они позволяют автоматически сгенерированному частичному типу предоставлять настраиваемые точки привязки для ручного написания кода, например:

```
partial class PaymentForm      // В автоматически сгенерированном файле
{
    ...
    partial void ValidatePayment (decimal amount);
}

partial class PaymentForm      // В написанном вручную файле
{
    ...
    partial void ValidatePayment (decimal amount)
    {
        if (amount > 100)
        ...
    }
}
```

Частичный метод состоит из двух частей: *определения* и *реализации*. Определение обычно записывается генератором кода, а реализация — вручную. Если реализация не предоставлена, тогда определение частичного метода при компиляции удаляется (вместе с кодом, где он вызывается). Это обеспечивает автоматически генерированному коду свободу в предоставлении точек привязки, не заставляя беспокоиться по поводу эффекта разбухания кода. Частичные методы должны иметь возвращаемый тип `void` и неявно являются `private`. Они не могут включать параметры `out`.

Расширенные частичные методы

Расширенные частичные методы (появившиеся в C# 9) предназначены для сценария обратной генерации кода, где программист определяет привязки, которые реализует генератор кода. Примером того, когда это происходит, могут служить *генераторы исходного кода* — средство Roslyn, позволяющее передать компилятору сборку, которая автоматически генерирует порции вашего кода.

Объявление частичного метода является *расширенным*, если оно начинается с модификатора доступности:

```
public partial class Test
{
    public partial void M1();           // Расширенный частичный метод
    private partial void M2();          // Расширенный частичный метод
}
```

Наличие модификатора доступности влияет не только на доступность, но заставляет компилятор трактовать объявление по-другому.

Расширенные частичные методы *обязаны* иметь реализации; они не исчезают, если не реализованы. В приведенном выше примере методы `M1` и `M2` должны иметь реализации, т.к. для каждого указаны модификаторы доступности (`public` и `private`).

Поскольку расширенные частичные методы не могут исчезнуть, они способны возвращать любой тип и включать параметры `out`:

```
public partial class Test
{
    public partial bool IsValid (string identifier);
    internal partial bool TryParse (string number, out int result);
}
```

Операция `nameof`

Операция `nameof` возвращает имя любого символа (типа, члена, переменной и т.д.) в виде строки:

```
int count = 123;
string name = nameof (count);      // name получает значение "count"
```

Преимущество использования операции `nameof` по сравнению с простым указанием строки связано со статической проверкой типов. Инструменты, подобные Visual Studio, способны воспринимать символические ссылки, поэтому переименование любого символа приводит к переименованию также всех ссылок на него.

Для указания имени члена типа, такого как поле или свойство, необходимо включать тип члена. Прием работает со статическими членами и членами экземпляра:

```
string name = nameof (StringBuilder.Length);
```

Результатом будет "Length". Чтобы возвратить "StringBuilder.Length", понадобится следующее выражение:

```
nameof (StringBuilder) + ". " + nameof (StringBuilder.Length);
```

Наследование

Класс может быть **унаследован** от другого класса с целью расширения или настройки первоначального класса. Наследование от класса позволяет повторно использовать функциональность данного класса вместо ее построения с нуля. Класс может наследоваться только от одного класса, но сам может быть унаследован множеством классов, формируя иерархию классов. В текущем примере мы начнем с определения класса по имени Asset:

```
public class Asset
{
    public string Name;
}
```

Далее мы определяем классы Stock и House, которые будут унаследованы от Asset. Классы Stock и House получат все, что имеет Asset, плюс любые дополнительные члены, которые в них будут определены:

```
public class Stock : Asset                                // унаследован от Asset
{
    public long SharesOwned;
}
public class House : Asset                               // унаследован от Asset
{
    public decimal Mortgage;
}
```

Вот как можно работать с данными классами:

```
Stock msft = new Stock { Name="MSFT",
                         SharesOwned=1000 };
Console.WriteLine (msft.Name);                          // MSFT
Console.WriteLine (msft.SharesOwned);                  // 1000

House mansion = new House { Name="Mansion",
                           Mortgage=250000 };
Console.WriteLine (mansion.Name);                      // Mansion
Console.WriteLine (mansion.Mortgage);                 // 250000
```

Производные классы Stock и House наследуют поле Name от базового класса Asset.



Производный класс также называют *подклассом*.

Базовый класс также называют *суперклассом*.

Полиморфизм

Ссылки *полиморфны*, т.е. переменная типа *x* может ссылаться на объект подкласса *x*. Например, рассмотрим следующий метод:

```
public static void Display (Asset asset)
{
    System.Console.WriteLine (asset.Name);
}
```

Метод *Display* способен отображать значение свойства *Name* объектов *Stock* и *House*, т.к. они оба являются *Asset*:

```
Stock msft      = new Stock ... ;
House mansion = new House ... ;

Display (msft);
Display (mansion);
```

В основе работы полиморфизма лежит тот факт, что подклассы (*Stock* и *House*) обладают всеми характеристиками своего базового класса (*Asset*). Однако обратное утверждение не будет верным. Если метод *Display* переписать так, чтобы он принимал *House*, то передавать ему *Asset* нельзя:

```
Display (new Asset());                                // Ошибка на этапе компиляции
public static void Display (House house)           // Asset приниматься не будет
{
    System.Console.WriteLine (house.Mortgage);
}
```

Приведение и ссылочные преобразования

Ссылка на объект может быть:

- неявно приведена вверх к ссылке на базовый класс;
- явно приведена вниз к ссылке на подкласс.

Приведение вверх и вниз между совместимыми ссылочными типами выполняет *ссылочное преобразование*: создается новая ссылка, которая указывает на *тот же самый* объект. Приведение вверх всегда успешно; приведение вниз успешно только в случае, когда объект надлежащим образом типизирован.

Приведение вверх

Операция приведения вверх создает ссылку на базовый класс из ссылки на подкласс. Например:

```
Stock msft = new Stock();
Asset a = msft;                                     // Приведение вверх
```

После приведения вверх переменная *a* по-прежнему ссылается на тот же самый объект *Stock*, что и переменная *msft*. Сам объект, на который имеются ссылки, не изменяется и не преобразуется:

```
Console.WriteLine (a == msft);                      // True
```

Несмотря на то что переменные `a` и `msft` ссылаются на один и тот же объект, переменная `a` обеспечивает более ограниченное представление этого объекта:

```
Console.WriteLine (a.Name);           // Нормально
Console.WriteLine (a.SharesOwned);     // Ошибка на этапе компиляции
```

Последняя строка кода вызывает ошибку на этапе компиляции, поскольку переменная `a` имеет тип `Asset`, хотя она ссылается на объект типа `Stock`. Чтобы получить доступ к полю `SharesOwned`, экземпляр `Asset` потребуется привести вниз к `Stock`.

Приведение вниз

Операция приведения вниз создает ссылку на подкласс из ссылки на базовый класс:

```
Stock msft = new Stock();
Asset a = msft;
Stock s = (Stock)a;           // Приведение вверх
Console.WriteLine (s.SharesOwned); // Ошибка не возникает
Console.WriteLine (s == a);       // True
Console.WriteLine (s == msft);    // True
```

Как и в случае приведения вверх, затрагиваются только ссылки, но не лежащий в основе объект. Приведение вниз должно указываться явно, потому что потенциально оно может не достичь успеха во время выполнения:

```
House h = new House();
Asset a = h;                      // Приведение вверх всегда успешно
Stock s = (Stock)a;               // Ошибка приведения вниз: a не является Stock
```

Когда приведение вниз терпит неудачу, генерируется исключение `InvalidCastException`. Это пример того, как работает проверка типов во время выполнения, которая более подробно рассматривается в разделе “Статическая проверка типов и проверка типов во время выполнения” далее в главе.

Операция as

Операция `as` выполняет приведение вниз, которое в случае неудачи вычисляется как `null` (вместо генерации исключения):

```
Asset a = new Asset();
Stock s = a as Stock;           // s равно null; исключение не генерируется
```

Операция удобна, когда нужно организовать последующую проверку результата на предмет `null`:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```



В отсутствие проверки подобного рода приведение удобно тем, что в случае его неудачи генерируется более полезное исключение. Мы можем проиллюстрировать сказанное, сравнив следующие две строки кода:

```
long shares = ((Stock)a).SharesOwned;    // Подход #1
long shares = (a as Stock).SharesOwned;    // Подход #2
```

Если `a` не является `Stock`, тогда первая строка кода сгенерирует исключение `InvalidOperationException`, которое обеспечит точное описание того, что пошло не так. Вторая строка кода сгенерирует исключение `NullReferenceException`, которое не даст однозначного ответа на вопрос, была ли переменная `a` не `Stock` или же просто `a` была равна `null`.

На описанную ситуацию можно взглянуть и по-другому: посредством операции приведения вы сообщаете компилятору о том, что *уверены* в типе заданного значения; если это не так, тогда в коде присутствует ошибка, а потому нужно сгенерировать исключение. С другой стороны, в случае операции `as` вы не уверены в типе значения и хотите организовать ветвление в соответствии с результатом во время выполнения.

Операция `as` не может выполнять специальные преобразования (см. раздел “Перегрузка операций” в главе 4), равно как и числовые преобразования:

```
long x = 3 as long; // Ошибка на этапе компиляции
```



Операция `as` и операции приведения также будут выполнять приведения вверх, хотя это не особенно полезно, поскольку всю необходимую работу сделает неявное преобразование.

Операция `is`

Операция `is` проверяет, соответствует ли переменная указанному шаблону. В C# поддерживается несколько видов шаблонов, наиболее важным из которых считается *шаблон типа*, где за ключевым словом `is` следует имя типа.

В таком контексте операция `is` проверяет, будет ли преобразование ссылки успешным — другими словами, является ли объект производным от указанного класса (или реализует ли он какой-то интерфейс). Операция `is` часто применяется для выполнения проверки перед приведением вниз:

```
if (a is Stock)
    Console.WriteLine (((Stock)a).SharesOwned);
```

Операция `is` также дает в результате `true`, если может успешно выполниться *распаковывающее преобразование* (см. раздел “Тип `object`” далее в главе). Тем не менее, она не принимает во внимание специальные или числовые преобразования.



Операция `is` работает со многими другими видами шаблонов, которые появились в последних версиях C#. За полным обсуждением обращайтесь в раздел “Шаблоны” главы 4.

Введение шаблонной переменной

Во время использования операции `is` можно вводить переменную:

```
if (a is Stock s)
    Console.WriteLine (s.SharesOwned);
```

Такой код эквивалентен следующему коду:

```
Stock s;
if (a is Stock)
{
    s = (Stock) a;
    Console.WriteLine (s.SharesOwned);
}
```

Введенная переменная доступна для “немедленного” потребления, поэтому показанный ниже код законен:

```
if (a is Stock s && s.SharesOwned > 100000)
    Console.WriteLine ("Wealthy");
```

И она остается в области видимости за пределами выражения `is`, давая возможность записывать так:

```
if (a is Stock s && s.SharesOwned > 100000)
    Console.WriteLine ("Wealthy");
else
    s = new Stock (); // s в области видимости
Console.WriteLine (s.SharesOwned); // s все еще в области видимости
```

Виртуальные функции-члены

Функция, помеченная как **виртуальная** (`virtual`), может быть *переопределена* в подклассах, где требуется предоставление ее специализированной реализации. Объявлять виртуальными можно методы, свойства, индексаторы и события:

```
public class Asset
{
    public string Name;
    public virtual decimal Liability => 0; // Свойство, сжатое до выражения
}
```

(Конструкция `Liability => 0` является сокращенной записью для `{ get { return 0; } }`. Дополнительные сведения о таком синтаксисе ищите в разделе “Свойства, сжатые до выражений” ранее в главе.)

Виртуальный метод переопределяется в подклассе с применением модификатора `override`:

```
public class Stock : Asset
{
    public long SharesOwned;
}
public class House : Asset
{
    public decimal Mortgage;
    public override decimal Liability => Mortgage;
}
```

По умолчанию свойство `Liability` класса `Asset` возвращает 0. Класс `Stock` не нуждается в специализации данного поведения. Однако класс `House` специализирует свойство `Liability`, чтобы возвращать значение `Mortgage`:

```
House mansion = new House { Name="McMansion", Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability);           // 250000
Console.WriteLine (a.Liability);                 // 250000
```

Сигнатуры, возвращаемые типы и доступность виртуального и переопределенного методов должны быть идентичными. Внутри переопределенного метода можно вызывать его реализацию из базового класса с помощью ключевого слова `base` (см. раздел “Ключевое слово `base`” далее в главе).



Вызов виртуальных методов внутри конструктора потенциально опасен, т.к. авторы подклассов при переопределении метода вряд ли знают о том, что работают с частично инициализированным объектом. Другими словами, переопределяемый метод может в итоге обращаться к методам или свойствам, зависящим от полей, которые пока еще не инициализированы конструктором.

Ковариантные возвращаемые типы

Начиная с версии C# 9, метод (или средство доступа `get` свойства) можно переопределять так, чтобы он возвращал *более производный* тип (более глубокий подкласс). Например:

```
public class Asset
{
    public string Name;
    public virtual Asset Clone() => new Asset { Name = Name };
}

public class House : Asset
{
    public decimal Mortgage;
    public override House Clone() => new House
        { Name = Name, Mortgage = Mortgage };
}
```

Поступать подобным образом разрешено, поскольку это не нарушает контракт о том, что метод `Clone` обязан возвращать экземпляр `Asset`: он возвращает экземпляр `House`, который *является* `Asset` (и многим другим).

До выхода версии C# 9 метод приходилось переопределять с идентичным возвращаемым типом:

```
public override Asset Clone() => new House { ... }
```

Подход по-прежнему работает, т.к. переопределенный метод `Clone` создает экземпляр `House`, а не `Asset`. Тем не менее, чтобы трактовать возвращенный объект как `House`, потребуется выполнить приведение вниз:

```
House mansion1 = new House { Name="McMansion", Mortgage=250000 };
House mansion2 = (House) mansion1.Clone();
```

Абстрактные классы и абстрактные члены

Экземпляры класса, объявленного как `abstract` (*абстрактный*), создавать не разрешено. Взамен можно создавать только экземпляры его конкретных *подклассов*. В абстрактных классах имеется возможность определять *абстрактные члены*. Абстрактные члены похожи на виртуальные члены за исключением того, что они не предоставляют стандартные реализации. Реализация должна обеспечиваться подклассом, если только подкласс тоже не объявлен как `abstract`:

```
public abstract class Asset
{
    // Обратите внимание на пустую реализацию
    public abstract decimal NetValue { get; }

    public class Stock : Asset
    {
        public long SharesOwned;
        public decimal CurrentPrice;
        // Переопределить подобно виртуальному методу
        public override decimal NetValue => CurrentPrice * SharesOwned;
    }
}
```

Скрытие унаследованных членов

В базовом классе и подклассе могут быть определены идентичные члены. Например:

```
public class A      { public int Counter = 1; }
public class B : A { public int Counter = 2; }
```

Говорят, что поле `Counter` в классе `B` *скрывает* поле `Counter` в классе `A`. Обычно это происходит случайно, когда член добавляется в базовый тип *после* того, как идентичный член был добавлен к подтипу. В таком случае компилятор генерирует предупреждение и затем разрешает неоднозначность следующим образом:

- ссылки на `A` (на этапе компиляции) привязываются к `A.Counter`;
- ссылки на `B` (на этапе компиляции) привязываются к `B.Counter`.

Иногда необходимо преднамеренно скрыть какой-то член; тогда к члену в подклассе можно применить ключевое слово `new`. Модификатор `new` *не делает ничего сверх того, что просто подавляет выдачу компилятором соответствующего предупреждения*:

```
public class A      { public      int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

Модификатор `new` сообщает компилятору — и другим программистам — о том, что дублирование члена произошло не случайно.



Ключевое слово `new` в языке C# перегружено и в разных контекстах имеет независимый смысл. В частности, операция `new` отличается от модификатора членов `new`.

Сравнение new и override

Рассмотрим следующую иерархию классов:

```
public class BaseClass
{
    public virtual void Foo() { Console.WriteLine ("BaseClass.Foo"); }

public class Overrider : BaseClass
{
    public override void Foo() { Console.WriteLine ("Overrider.Foo"); }

public class Hider : BaseClass
{
    public new void Foo() { Console.WriteLine ("Hider.Foo"); }
```

Ниже приведен код, демонстрирующий отличия в поведении классов Overrider и Hider:

```
Overrider over = new Overrider();
BaseClass b1 = over;
over.Foo();           // Overrider.Foo
b1.Foo();            // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();             // Hider.Foo
b2.Foo();            // BaseClass.Foo
```

Запечатывание функций и классов

С помощью ключевого слова **sealed** переопределенная функция может запечатывать свою реализацию, предотвращая ее переопределение другими подклассами. В ранее показанном примере виртуальной функции-члена можно было бы запечатать реализацию **Liability** в классе **House**, чтобы запретить переопределение **Liability** в классе, производном от **House**:

```
public sealed override decimal Liability { get { return Mortgage; } }
```

Модификатор **sealed** можно также применить к самому классу, чтобы предотвратить создание его подклассов. Запечатывание класса встречается чаще, чем запечатывание функции-члена.

Функцию-член можно запечатывать с целью запрета ее переопределения, но не скрытия.

Ключевое слово **base**

Ключевое слово **base** похоже на ключевое слово **this**. Оно служит двум важным целям:

- доступ к функции-члену базового класса при ее переопределении в подклассе;
- вызов конструктора базового класса (см. следующий раздел).

В приведенном ниже примере ключевое слово `base` в классе `House` используется для доступа к реализации `Liability` из `Asset`:

```
public class House : Asset
{
    ...
    public override decimal Liability => base.Liability + Mortgage;
}
```

С помощью ключевого слова `base` мы получаем доступ к свойству `Liability` класса `Asset` *невиртуальным* способом. Это значит, что мы всегда обращаемся к версии данного свойства из `Asset` независимо от действительного типа экземпляра во время выполнения.

Тот же самый подход работает в ситуации, когда `Liability` скрывается, а не переопределяется. (Получить доступ к скрытым членам можно также путем приведения к базовому классу перед обращением к члену.)

Конструкторы и наследование

В подклассе должны быть объявлены собственные конструкторы. Конструкторы базового класса *доступны* в производном классе, но они никогда автоматически не *наследуются*. Например, если классы `Baseclass` и `Subclass` определены следующим образом:

```
public class Baseclass
{
    public int X;
    public Baseclass () { }
    public Baseclass (int x) => this.X = x;
}

public class Subclass : Baseclass { }
```

то приведенный ниже код будет недопустимым:

```
Subclass s = new Subclass (123);
```

Следовательно, в классе `Subclass` должны быть “повторно определены” любые конструкторы, к которым необходимо открыть доступ. Тем не менее, с применением ключевого слова `base` можно вызывать любой конструктор базового класса:

```
public class Subclass : Baseclass
{
    public Subclass (int x) : base (x) { }
```

Ключевое слово `base` работает подобно ключевому слову `this`, но только вызывает конструктор базового класса.

Конструкторы базового класса всегда выполняются первыми, гарантируя тем самым, что базовая инициализация произойдет перед *специализированной* инициализацией.

Неявный вызов конструктора без параметров базового класса

Если в конструкторе подкласса опустить ключевое слово `base`, то будет неявно вызываться конструктор без параметров базового класса:

```
public class BaseClass
{
    public int X;
    public BaseClass() { X = 1; }

}

public class Subclass : BaseClass
{
    public Subclass() { Console.WriteLine (X); } // 1
}
```

Если базовый класс не имеет доступного конструктора без параметров, тогда в конструкторах подклассов придется использовать ключевое слово `base`. Это означает, что базовый класс с (одним лишь) конструктором, принимающим несколько параметров, обязывает подклассы вызывать его:

```
class Baseclass
{
    public Baseclass (int x, int y, int z, string s, DateTime d) { ... }

}

public class Subclass : Baseclass
{
    public Subclass (int x, int y, int z, string s, DateTime d)
        : base (x, y, z, s, d) { ... }
}
```

Обязательные члены (C# 11)

Требование к подклассам вызывать конструктор базового класса может стать обременительным в крупных иерархиях классов, если имеется много конструкторов с многочисленными параметрами. Иногда лучшее решение — вообще избегать конструкторов и полагаться исключительно на инициализаторы объектов для установки полей или свойств во время создания экземпляров. Для содействия в этом, начиная с версии C# 11, поле или свойство можно пометить как `required` (обязательное):

```
public class Asset
{
    public required string Name;
}
```

Обязательный член должен быть заполнен через инициализатор объекта при конструировании:

```
Asset a1 = new Asset { Name="House" };           // Нормально
Asset a2 = new Asset();                          // Ошибка: не скомпилируется!
```

Если необходимо также реализовать конструктор, тогда за счет применения атрибута `[SetsRequiredMembers]` можно обойти ограничение обязательного члена для данного конструктора:

```

public class Asset
{
    public required string Name;
    public Asset() { }

    [System.Diagnostics.CodeAnalysis.SetsRequiredMembers]
    public Asset (string n) => Name = n;
}

```

Потребители теперь могут извлечь выгоду из удобства этого конструктора без каких-либо компромиссов:

```

Asset a1 = new Asset { Name = "House" };      // Нормально
Asset a2 = new Asset ("House");                // Нормально
Asset a3 = new Asset();                        // Ошибка!

```

Обратите внимание, что здесь также определен конструктор без параметров (для использования с инициализатором объекта). Его присутствие также гарантирует, что в подклассах не придется воспроизводить любой конструктор. В следующем примере принято решение не реализовывать в классе House удобный конструктор:

```

public class House : Asset { }                  // Нет конструктора - нет забот!
House h1 = new House { Name = "House" };        // Нормально
House h2 = new House();                         // Ошибка!

```

Конструктор и порядок инициализации полей

Когда объект создан, инициализация происходит в указанном ниже порядке.

От подкласса к базовому классу:

- инициализируются поля;
- вычисляются аргументы для вызова конструкторов базового класса.

От базового класса к подклассу:

- выполняются тела конструкторов.

Порядок демонстрируется в следующем коде:

```

public class B
{
    int x = 1;           // Выполняется третьим
    public B (int x)
    {
        ...              // Выполняется четвертым
    }
}

public class D : B
{
    int y = 1;           // Выполняется первым
    public D (int x)
        : base (x + 1) // Выполняется вторым
    {
        ...              // Выполняется пятым
    }
}

```

Наследование от классов с основными конструкторами

Подклассы из классов с основными конструкторами можно создавать с помощью следующего синтаксиса:

```
public class Baseclass (int x) { ... }
public class Subclass (int x, int y) : Baseclass (x) { ... }
```

Вызов `Baseclass (x)` эквивалентен вызову `base (x)` в показанном ниже примере:

```
public class Subclass : Baseclass
{
    public Subclass (int x, int y) : base (x) { ... }
}
```

Перегрузка и распознавание

Наследование оказывает интересное влияние на перегрузку методов. Предположим, что есть две перегруженные версии метода:

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

При вызове перегруженной версии приоритет получает наиболее специфичный тип:

```
House h = new House (...);
Foo(h);                                // Вызывается Foo(House)
```

Конкретная перегруженная версия, подлежащая вызову, определяется статически (на этапе компиляции), а не во время выполнения. В показанном ниже коде вызывается `Foo(Asset)` несмотря на то, что типом времени выполнения переменной `a` является `House`:

```
Asset a = new House (...);
Foo(a);                                // Вызывается Foo(Asset)
```



Если привести `Asset` к `dynamic` (см. главу 4), тогда решение о том, какая перегруженная версия должна вызываться, откладывается до стадии выполнения, и выбор будет основан на действительном типе объекта:

```
Asset a = new House (...);
Foo ((dynamic)a);                      // Вызывается Foo(House)
```

Тип `object`

Тип `object` (`System.Object`) представляет собой первоначальный базовый класс для всех типов. Любой тип может быть неявно приведен вверх к `object`.

Чтобы проиллюстрировать, насколько это полезно, рассмотрим универсальный стек. Стек является структурой данных, работа которой основана на принципе LIFO ("last in, first out" — "последним пришел, первым обслужен"). Стек поддерживает две операции: *помещение* объекта в стек и *извлечение* объекта из

стека. Ниже показана простая реализация, которая способна хранить до 10 объектов:

```
public class Stack
{
    int position;
    object[] data = new object[10];
    public void Push (object obj)    { data[position++] = obj; }
    public object Pop()              { return data[--position]; }
}
```

Поскольку `Stack` работает с типом `object`, методы `Push` и `Pop` класса `Stack` можно применять с экземплярами *любого типа*:

```
Stack stack = new Stack();
stack.Push ("sausage");
string s = (string) stack.Pop(); // Приведение вниз должно быть явным
Console.WriteLine (s);          // Выводит sausage
```

Тип `object` относится к ссылочным типам в силу того, что представляет собой класс. Несмотря на данное обстоятельство, типы значений, такие как `int`, также можно приводить к `object`, а `object` приводить к ним, и потому они могут быть помещены в стек. Такая особенность C# называется *унификацией типов* и демонстрируется ниже:

```
stack.Push (3);
int three = (int) stack.Pop();
```

Когда запрашивается приведение между типом значения и типом `object`, среда CLR должна выполнить специальную работу по преодолению семантических отличий между типами значений и ссылочными типами. Процессы называются *упаковкой* (*boxing*) и *распаковкой* (*unboxing*).



В разделе “Обобщения” далее в главе будет показано, как усовершенствовать класс `Stack`, чтобы улучшить поддержку стеков однотипных элементов.

Упаковка и распаковка

Упаковка представляет собой действие по приведению экземпляра типа значения к экземпляру ссылочного типа. Ссылочным типом может быть либо класс `object`, либо интерфейс (см. раздел “Интерфейсы” далее в главе)¹. В следующем примере мы упаковываем `int` в `object`:

```
int x = 9;
object obj = x;           // Упаковать int
```

Распаковка является обратной операцией, которая предусматривает приведение объекта к исходному типу значения:

```
int y = (int)obj;        // Распаковать int
```

¹ Ссылочным типом может также быть `System.ValueType` или `System.Enum` (см. главу 6).

Распаковка требует явного приведения. Исполняющая среда проверяет, соответствует ли указанный тип значения действительному объектному типу, и генерирует исключение `InvalidCastException`, если это не так. Например, показанный далее код приведет к генерации исключения, поскольку `long` не точно соответствует `int`:

```
object obj = 9;           // Для значения 9 выводится тип int
long x = (long) obj;      // Генерируется исключение
InvalidCastException
```

Однако показанный далее код выполняется успешно:

```
object obj = 9;
long x = (int) obj;
```

Следующий код также не вызывает ошибок:

```
object obj = 3.5;          // Для значения 3.5 выводится тип double
int x = (int) (double) obj; // x теперь равно 3
```

В последнем примере (`double`) осуществляет *распаковку*, после чего (`int`) выполняет *числовое преобразование*.



Упаковывающие преобразования критически важны в обеспечении унифицированной системы типов. Тем не менее, эта система не идеальна: в разделе “Обобщения” далее в главе будет показано, что *вариантность массивов и обобщений* поддерживает только *ссыльные преобразования*, но не *упаковывающие преобразования*:

```
object[] a1 = new string[3]; // Допустимо
object[] a2 = new int[3];    // Ошибка
```

Семантика копирования при упаковке и распаковке

Упаковка *копирует* экземпляр типа значения в новый объект, а распаковка *копирует* содержимое данного объекта обратно в экземпляр типа значения. В приведенном ниже примере изменение значения `i` не вызывает изменения ранее упакованной копии:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed);           // 3
```

Статическая проверка типов и проверка типов во время выполнения

Программы на языке C# подвергаются проверке типов как статически (на этапе компиляции), так и во время выполнения (средой CLR).

Статическая проверка типов позволяет компилятору контролировать корректность программы, не запуская ее. Показанный ниже код не скомпилируется, т.к. компилятор принудительно применяет статическую проверку типов:

```
int x = "5";
```

Проверка типов во время выполнения осуществляется средой CLR, когда происходит приведение вниз через ссылочное преобразование или распаковку:

```
object y = "5";
int z = (int) y; // Ошибка времени выполнения, неудача приведения вниз
```

Проверка типов во время выполнения возможна по той причине, что каждый объект в куче внутренне хранит небольшой маркер типа. Данный маркер может быть извлечен посредством вызова метода `GetType` класса `object`.

Метод `GetType` и операция `typeof`

Все типы в C# во время выполнения представлены с помощью экземпляра `System.Type`. Получить объект `System.Type` можно двумя основными путями:

- вызвать метод `GetType` на экземпляре;
- воспользоваться операцией `typeof` на имени типа.

Результат `GetType` вычисляется во время выполнения, а `typeof` — статически на этапе компиляции (когда вовлечены параметры обобщенных типов, они распознаются компилятором JIT).

В классе `System.Type` предусмотрены свойства для имени типа, сборки, базового типа и т.д.:

```
Point p = new Point();
Console.WriteLine (p.GetType().Name);                                // Point
Console.WriteLine (typeof (Point).Name);                                // Point
Console.WriteLine (p.GetType() == typeof(Point));                      // True
Console.WriteLine (p.X.GetType().Name);                                 // Int32
Console.WriteLine (p.Y.GetType().FullName);                            // System.Int32
public class Point { public int X, Y; }
```

В `System.Type` также имеются методы, которые действуют в качестве шлюза для модели рефлексии времени выполнения, которая описана в главе 18.

Метод `ToString`

Метод `ToString` возвращает стандартное текстовое представление экземпляра типа. Данный метод переопределяется всеми встроенными типами. Ниже приведен пример применения метода `ToString` типа `int`:

```
int x = 1;
string s = x.ToString();                                              // s равно "1"
```

Переопределять метод `ToString` в специальных типах можно следующим образом:

```
Panda p = new Panda { Name = "Petey" };
Console.WriteLine (p);                                                 // Petey
public class Panda
{
    public string Name;
    public override string ToString() => Name;
}
```

Если метод `ToString` не переопределять, то он будет возвращать имя типа.



В случае вызова *переопределенного* члена класса `object`, такого как `ToString`, непосредственно на типе значения упаковка не производится. Упаковка происходит позже, только если осуществляется приведение:

```
int x = 1;
string s1 = x.ToString();      // Вызывается на неупакованном значении
object box = x;
string s2 = box.ToString();    // Вызывается на упакованном значении
```

Список членов `object`

Ниже приведен список всех членов `object`:

```
public class Object
{
    public Object();
    public extern Type GetType();
    public virtual bool Equals (object obj);
    public static bool Equals (object objA, object objB);
    public static bool ReferenceEquals (object objA, object objB);
    public virtual int GetHashCode();
    public virtual string ToString();
    protected virtual void Finalize();
    protected extern object MemberwiseClone();
}
```

Методы `Equals`, `ReferenceEquals` и `GetHashCode` будут описаны в разделе “Сравнение эквивалентности” главы 6.

Структуры

Структура похожа на класс, но обладает указанными ниже ключевыми отличиями.

- Структура является типом значения, тогда как класс — ссылочным типом.
- Структура не поддерживает наследование (за исключением того, что она неявно порождена от `object`, а точнее — от `System.ValueType`).

Структура способна иметь все те же члены, что и класс, за исключением финализатора. И поскольку создавать подклассы для нее невозможно, члены не могут быть помечены как `virtual`, `abstract` или `protected`.



До выхода версии C# 10 в структурах было запрещено определять инициализаторы полей и конструкторы без параметров. Хотя сейчас этот запрет смягчен — в первую очередь в пользу структур типа записей (см. раздел “Записи” в главе 4), — стоит тщательно подумать, прежде чем определять такие конструкции, потому что они могут привести к путанице в поведении, как объясняется в разделе “Семантика конструирования структур” далее в главе.

Структура подходит там, где желательно иметь семантику типа значения. Хорошими примерами могут служить числовые типы, для которых более естественным способом присваивания будет копирование значения, а не ссылки. Поскольку структура представляет собой тип значения, каждый экземпляр не требует создания объекта в куче; это дает ощутимую экономию при создании большого количества экземпляров типа. Например, создание массива с элементами типа значения требует только одного выделения памяти в куче.

Из-за того, что структуры являются типами значений, экземпляр не может быть равен `null`. Стандартное значение для структуры — пустой экземпляр со всеми пустыми полями (установленными в свои стандартные значения).

Семантика конструирования структур



До выхода версии C# 11 каждому полю в структуре должно было быть явно присвоено значение в конструкторе (или в инициализаторе поля). Сейчас это ограничение смягчено.

Стандартный конструктор

В дополнение к любым определяемым вами конструкторам структура всегда имеет неявный конструктор без параметров, который выполняет побитовое обнуление полей структуры (устанавливая для них стандартные значения):

```
Point p = new Point();           // p.x и p.y получат значение 0
struct Point { int x, y; }
```

Даже если вы определяете собственный конструктор без параметров, то неявный конструктор без параметров по-прежнему существует, и к нему можно получить доступ через ключевое слово `default`:

```
Point p1 = new Point();           // p1.x и p1.y получат значение 1
Point p2 = default;              // p2.x и p2.y получат значение 0
struct Point
{
    int x = 1;
    int y;
    public Point() => y = 1;
}
```

В приведенном примере поле `x` инициализируется значением 1 с помощью инициализатора поля, а поле `y` — значением 1 через конструктор без параметров. И все же с использованием ключевого слова `default` мы смогли создать экземпляр `Point`, который обходит обе инициализации. Доступ к стандартному конструктору можно получить и другими способами, как демонстрируется в следующем примере:

```
var points = new Point[10]; // Все точки в массиве будут иметь значение (0,0)
var test = new Test();     // test.p будет иметь значение (0,0)
class Test { Point p; }
```



Наличие двух конструкторов без параметров может стать источником путаницы и, возможно, является хорошей причиной избегать определения инициализаторов полей и явных конструкторов без параметров в структурах.

Хорошая стратегия в отношении структур предусматривает их проектирование так, чтобы стандартное значение было допустимым состоянием, делая инициализацию избыточной. Например, вместо инициализации свойства следующим образом:

```
public string Protocol { get; set; } = "https";
```

можно поступить так:

```
struct WebOptions
{
    string protocol;
    public string Protocol { get => protocol ?? "https";
                           set => protocol = value; }
}
```

Структуры и функции, поддерживающие только чтение

К структуре можно применять модификатор `readonly`, чтобы все поля в ней были `readonly`; такой прием помогает заявить о своем намерении и предоставляет компилятору большую свободу в плане оптимизации:

```
readonly struct Point
{
    public readonly int X, Y; // X и Y обязаны быть readonly
}
```

На тот случай, когда модификатор `readonly` необходимо применять на более детализированном уровне, в C# 8 появилась возможность применять его к функциям структуры, гарантируя тем самым, что если функция попытается модифицировать любое поле, то генерируется ошибка на этапе компиляции:

```
struct Point
{
    public int X, Y;
    public readonly void ResetX() => X = 0; // Ошибка!
}
```

Если функция `readonly` вызывает функцию не `readonly`, тогда компилятор генерирует предупреждение (и защитным образом копирует структуру, чтобы устранить возможность ее изменения).

Сылочные структуры



Сылочные структуры были введены в C# 7.2 как нишевое средство в интересах структур `Span<T>` и `ReadOnlySpan<T>`, которые будут рассматриваться в главе 23 (и высокооптимизированной структуре `Utf8JsonReader`, описанной в главе 11). Упомянутые структуры оказывают содействие технологии микрооптимизации, которая направлена на сокращение выделения памяти.

В отличие от ссылочных типов, экземпляры которых всегда размещаются в куче, типы значений находятся *на месте* (где была объявлена переменная). Если тип значения относится к параметру или локальной переменной, то размещение произойдет в стеке:

```
void SomeMethod()
{
    Point p; // p будет находиться в стеке
}

struct Point { public int X, Y; }
```

Но если тип значения относится к полю в классе, тогда размещение произойдет в куче:

```
class MyClass
{
    Point p; //Находится в куче, поскольку экземпляры MyClass размещаются в куче
}
```

Аналогичным образом массивы структур находятся в куче, и упаковка структуры приводит к ее размещению в куче.

Добавление модификатора **ref** к объявлению структуры гарантирует, что она сможет размещаться только в стеке. Попытка использования *ссылочной структуры* таким способом, чтобы она могла располагаться в куче, приводит к генерации ошибки на этапе компиляции:

```
var points = new Point [100];           // Ошибка: не скомпилируется!
ref struct Point { public int X, Y; }
class MyClass { Point P; } // Ошибка: не скомпилируется!
```

Ссылочные структуры были введены главным образом в интересах структур **Span<T>** и **ReadOnlySpan<T>**. Поскольку экземпляры **Span<T>** и **ReadOnlySpan<T>** могут существовать только в стеке, у них есть возможность безопасно содержать в себе память, выделенную в стеке.

Ссылочные структуры не могут принимать участие в каком-либо средстве C#, которое прямо или косвенно привносит вероятность существования в куче. Сюда входит несколько расширенных средств C#, которые рассматриваются в главе 4, в частности лямбда-выражения, итераторы и асинхронные функции (потому что все упомянутые средства “за кулисами” создают классы с полями). Кроме того, ссылочные структуры не могут находиться внутри нессылочных структур и не могут реализовывать интерфейсы (т.к. это может приводить к упаковке).

Модификаторы доступа

Для содействия инкапсуляции тип или член типа может ограничивать свою доступность другим типам и сборкам за счет добавления к объявлению одного из описанных ниже *модификаторов доступа*.

public

Полная доступность. Это явная доступность для членов перечисления либо интерфейса.

internal

Доступность только внутри содержащей сборки или в дружественных сборках. Это стандартная доступность для невложенных типов.

private

Доступность только внутри содержащего типа. Это стандартная доступность для членов класса или структуры.

protected

Доступность только внутри содержащего типа или в его подклассах.

protected internal

Объединение доступности **protected** и **internal**. Член **protected internal** доступен двумя путями.

private protected

Пересечение доступности **protected** и **internal**. Член **private protected** доступен только внутри содержащего типа или в подклассах, которые находятся в той же самой сборке (что делает его менее доступным, чем **protected** либо **internal** по отдельности).

file (начиная с версии C# 11)

Доступность только внутри того же самого файла. Предназначен для использования генераторами исходного кода (см. раздел “Расширенные частичные методы” ранее в главе). Данный модификатор можно применять только к объявлениям типов.

Примеры

Класс Class2 доступен извне его сборки; Class1 — нет:

```
class Class1 {} // Class1 является internal (по умолчанию)
public class Class2 {}
```

Класс ClassB открывает поле x другим типам в той же сборке; ClassA — нет:

```
class ClassA { int x; } // x является private (по умолчанию)
class ClassB { internal int x; }
```

Функции внутри Subclass могут вызывать Bar, но не Foo:

```
class BaseClass {
    void Foo() {} // Foo является private (по умолчанию)
    protected void Bar() {}
}
```

```
class Subclass : BaseClass
{
    void Test1() { Foo(); }           // Ошибка - доступ к Foo невозможен
    void Test2() { Bar(); }          // Нормально
}
```

Дружественные сборки

Члены `internal` можно открывать другим *дружественным* сборкам, добавляя атрибут сборки `System.Runtime.CompilerServices.InternalVisibleTo`, в котором указано имя дружественной сборки:

```
[assembly: InternalVisibleTo ("Friend")]
```

Если дружественная сборка имеет строгое имя (см. главу 17), тогда потребуется указать ее *полный* 160-байтовый открытый ключ:

```
[assembly: InternalVisibleTo ("StrongFriend, PublicKey=0024f000048c...")]
```

Извлечь полный открытый ключ из строго именованной сборки можно с помощью запроса LINQ (более детально LINQ рассматривается в главе 8):

```
string key = string.Join ("",
    Assembly.GetExecutingAssembly().GetName ().GetPublicKey ()
    .Select (b => b.ToString ("x2")));
```



В сопровождающем книгу примере для LINQPad предлагается выбрать сборку и затем скопировать полный открытый ключ сборки в буфер обмена.

Установление верхнего предела доступности

Тип устанавливает верхний предел доступности объявленных в нем членов. Наиболее распространенным примером такого установления является ситуация, когда есть тип `internal` с членами `public`. В качестве примера взгляните на следующий код:

```
class C { public void Foo() {} }
```

Стандартная доступность `internal` класса `C` устанавливает верхний предел доступности метода `Foo`, по существу делая `Foo` внутренним. Распространенная причина пометки `Foo` как `public` связана с облегчением рефакторинга, если позже будет решено изменить доступность класса `C` на `public`.

Ограничения, накладываемые на модификаторы доступа

При переопределении метода из базового класса доступность должна быть идентичной доступности переопределяемого метода. Например:

```
class BaseClass          { protected virtual void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} } // Нормально
class Subclass2 : BaseClass { public override void Foo() {} } // Ошибка
```

(Исключением является случай переопределения метода `protected internal` в другой сборке, при котором переопределяемый метод должен быть просто `protected`.)

Компилятор предотвращает несогласованное использование модификаторов доступа. Например, подкласс может иметь меньшую доступность, чем базовый класс, но не большую:

```
internal class A {}
public class B : A {} // Ошибка
```

Интерфейсы

Интерфейс похож на класс, но он только задает *поведение* и не хранит состояние (данные). Интерфейс обладает следующими особенностями.

- В интерфейсе можно определять только функции, но не поля.
- Все члены интерфейса *неявно абстрактные*. (Из этого правила существуют исключения, которые будут описаны в разделах “Стандартные члены интерфейса” и “Статические члены интерфейса” далее в главе.)
- Класс (или структура) может реализовывать *несколько* интерфейсов. Напротив, класс может быть унаследован только от *одного* класса, а структура вообще не поддерживает наследование (за исключением того, что она порождена от `System.ValueType`).

Объявление интерфейса похоже на объявление класса, но интерфейс (обычно) не предоставляет никакой реализации для своих членов, т.к. они неявно абстрактные. Члены интерфейса будут реализованы классами и структурами, которые реализуют данный интерфейс. Интерфейс может содержать только функции, т.е. методы, свойства, события и индексаторы (что неслучайно в точности соответствует членам класса, которые могут быть абстрактными).

Ниже показано определение интерфейса `IEnumerator` из пространства имен `System.Collections`:

```
public interface IEnumarator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

Члены интерфейса всегда неявно являются `public`, и для них нельзя объявлять какие-либо модификаторы доступа. Реализация интерфейса означает предоставление реализации `public` для всех его членов:

```
internal class Countdown : IEnumarator
{
    int count = 11;
    public bool MoveNext() => count-- > 0;
    public object Current => count;
    public void Reset() { throw new NotSupportedException(); }
}
```

Объект можно неявно приводить к любому интерфейсу, который он реализует:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
    Console.WriteLine(e.Current); // 109876543210
```



Несмотря на то что `Countdown` — внутренний класс, его члены, которые реализуют интерфейс `IEnumerator`, могут открыто вызываться за счет приведения экземпляра `Countdown` к `IEnumerator`. Скажем, если какой-то открытый тип в той же сборке определяет метод следующим образом:

```
public static class Util
{
    public static object GetCountDown() => new CountDown();
}
```

то в вызывающем коде внутри другой сборки можно поступать так:

```
IEnumerator e = (IEnumerator) Util.GetCountDown();
e.MoveNext();
```

Если бы сам интерфейс `IEnumerator` был определен как `internal`, тогда подобное оказалось бы невозможным.

Расширение интерфейса

Интерфейсы могут быть производными от других интерфейсов. Вот пример:

```
public interface IUndoable { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); }
```

Интерфейс `IRedoable` “наследует” все члены интерфейса `IUndoable`. Другими словами, типы, которые реализуют `IRedoable`, обязаны также реализовывать члены `IUndoable`.

Явная реализация членов интерфейса

Реализация множества интерфейсов временами может приводить к конфликтам между сигнатурами членов. Разрешать такие конфликты можно за счет явной реализации члена интерфейса. Рассмотрим следующий пример:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
    public void Foo()
    {
        Console.WriteLine("Widget's implementation of I1.Foo");
        // Реализация I1.Foo в Widget
    }
}
```

```

int I2.Foo()
{
    Console.WriteLine ("Widget's implementation of I2.Foo");
    // Реализация I2.Foo в Widget
    return 42;
}
}

```

Поскольку интерфейсы I1 и I2 имеют методы Foo с конфликтующими сигнатурами, метод Foo интерфейса I2 в классе Widget реализуется явно, что позволяет двум методам сосуществовать в рамках одного класса. Единственный способ вызова явно реализованного метода предусматривает приведение к его интерфейсу:

```

Widget w = new Widget();
w.Foo();                                // Реализация I1.Foo в Widget
((I1)w).Foo();                            // Реализация I1.Foo в Widget
((I2)w).Foo();                            // Реализация I2.Foo в Widget

```

Еще одной причиной явной реализации членов интерфейса может быть необходимость скрытия членов, которые являются узкоспециализированными и нарушающими нормальный сценарий использования типа. Скажем, тип, который реализует ISerializable, обычно будет избегать демонстрации членов ISerializable, если только не осуществляется явное приведение к упомянутому интерфейсу.

Реализация виртуальных членов интерфейса

Невидимо реализованный член интерфейса по умолчанию будет запечатанным. Чтобы его можно было переопределить, он должен быть помечен в базовом классе как `virtual` или `abstract`. Например:

```

public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
    public virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox
{
    public override void Undo() => Console.WriteLine ("RichTextBox.Undo");
}

```

Обращение к такому члену интерфейса либо через базовый класс, либо через интерфейс приводит к вызову его реализации из подкласса:

```

RichTextBox r = new RichTextBox();
r.Undo();                                // RichTextBox.Undo
((IUndoable)r).Undo();                    // RichTextBox.Undo
((TextBox)r).Undo();                      // RichTextBox.Undo

```

Явно реализованный член интерфейса не может быть помечен как `virtual`, равно как и не может быть переопределен обычным образом. Однако он может быть *реализован повторно*.

Повторная реализация члена интерфейса в подклассе

Подкласс может повторно реализовывать любой член интерфейса, который уже реализован базовым классом. Повторная реализация перехватывает реализацию члена (при вызове через интерфейс) и работает вне зависимости от того, является ли член виртуальным в базовом классе. Повторная реализация также работает в ситуации, когда член реализован неявно или явно — хотя, как будет продемонстрировано, в последнем случае она работает лучше.

В показанном ниже примере класс TextBox явно реализует `IUndoable.Undo`, а потому данный метод не может быть помечен как `virtual`. Чтобы его “переопределить”, класс RichTextBox обязан повторно реализовать метод Undo интерфейса `IUndoable`:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
    void IUndoable.Undo() => Console.WriteLine ("TextBox.Undo");

    public class RichTextBox : TextBox, IUndoable
    {
        public void Undo() => Console.WriteLine ("RichTextBox.Undo");
    }
}
```

Обращение к повторно реализованному методу через интерфейс приводит к вызову его реализации из подкласса:

```
RichTextBox r = new RichTextBox();
r.Undo();                                // RichTextBox.Undo Случай 1
((IUndoable)r).Undo();      // RichTextBox.Undo Случай 2
```

При том же самом определении RichTextBox предположим, что TextBox реализует метод Undo *неявно*:

```
public class TextBox : IUndoable
{
    public void Undo() => Console.WriteLine ("TextBox.Undo");
}
```

В итоге мы имеем еще один способ вызова метода Undo, который “нарушает” систему, как показано в случае 3:

```
RichTextBox r = new RichTextBox();
r.Undo();                                // RichTextBox.Undo Случай 1
((IUndoable)r).Undo();      // RichTextBox.Undo Случай 2
((TextBox)r).Undo();      // TextBox.Undo Случай 3
```

Случай 3 демонстрирует тот факт, что перехват повторной реализации результативен, только когда член вызывается через интерфейс, а не через базовый класс. Обычно подобное нежелательно, т.к. может означать несогласованную семантику. Это делает повторную реализацию наиболее подходящей в качестве стратегии для переопределения явно реализованных членов интерфейса.

Альтернативы повторной реализации членов интерфейса

Даже при явной реализации членов повторная реализация проблематична по следующим причинам.

- Подкласс не имеет возможности вызывать метод базового класса.
- Автор базового класса мог не предполагать, что метод будет повторно реализован, и потому не учел потенциальные последствия.

Повторная реализация может оказаться хорошим последним средством в ситуации, когда создание подклассов не предвиделось. Тем не менее, лучше проектировать базовый класс так, чтобы потребность в повторной реализации никогда не возникала. Данной цели можно достичь двумя путями.

- В случае неявной реализации члена пометьте его как `virtual`, если подобное возможно.
- В случае явной реализации члена используйте следующий шаблон, если предполагается, что в подклассах может понадобиться переопределение любой логики:

```
public class TextBox : IUndoable
{
    void IUndoable.Undo() => Undo(); // Вызывает метод, определенный ниже
    protected virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}
public class RichTextBox : TextBox
{
    protected override void Undo() => Console.WriteLine("RichTextBox.Undo");
}
```

Если создание подклассов не предвидится, тогда класс можно пометить как `sealed`, чтобы предотвратить повторную реализацию членов интерфейса.

Интерфейсы и упаковка

Преобразование структуры в интерфейс приводит к упаковке. Обращение к неявно реализованному члену структуры упаковку не вызывает:

```
interface I { void Foo(); }
struct S : I { public void Foo() {} }
...
S s = new S();
s.Foo(); // Упаковка не происходит
I i = s; // Упаковка происходит во время приведения к интерфейсу
i.Foo();
```

Стандартные члены интерфейса

Начиная с версии C# 8, к члену интерфейса можно добавлять стандартную реализацию, делая его необязательным для реализации:

```
interface ILogger
{
    void Log (string text) => Console.WriteLine (text);
}
```

Такая возможность полезна, когда необходимо добавить член к интерфейсу, который определен в популярной библиотеке, не нарушая работу (потенциально многих тысяч) реализаций.

Стандартные реализации всегда явные, так что если класс, реализующий ILogger, не определит метод Log, то вызывать его можно будет только через интерфейс:

```
class Logger : ILogger { }  
...  
(ILogger)new Logger()).Log ("message");
```

Это предотвращает проблему наследования множества реализаций: если тот же самый стандартный член добавлен в два интерфейса, которые реализует класс, то никогда не возникнет неоднозначность относительно того, какой член вызывать.

Статические члены интерфейса

В интерфейсах можно также объявлять статические члены. Есть два вида статических членов интерфейсов:

- статические невиртуальные члены интерфейсов;
- статические виртуальные/абстрактные члены интерфейсов.



В отличие от членов *экземпляра* статические члены интерфейсов по умолчанию не являются виртуальными. Чтобы сделать статический член интерфейса виртуальным, его необходимо пометить как static abstract или static virtual.

Статические невиртуальные члены интерфейсов

Статические невиртуальные члены интерфейсов существуют главным образом для того, чтобы облегчить написание стандартных членов интерфейсов. Они не реализуются классами или структурами, а взамен потребляются напрямую. Наряду с методами, свойствами, событиями и индексаторами статические невиртуальные члены разрешают использовать поля, доступ к которым обычно осуществляется из кода внутри стандартных реализаций членов:

```
interface ILogger  
{  
    void Log (string text) =>  
        Console.WriteLine (Prefix + text);  
    static string Prefix = "";  
}
```

Невиртуальные члены интерфейсов по умолчанию являются открытыми, так что к ним всегда можно обращаться извне:

```
	ILogger.Prefix = "File log: ";
```

Вы можете ограничить это, добавив модификатор доступности (такой как `private`, `protected` или `internal`).

Поля экземпляра (по-прежнему) запрещены, что согласуется с принципом интерфейсов, который заключается в том, что интерфейсы определяют *поведение*, но не *состояние*.

Статические виртуальные/абстрактные члены интерфейсов

Статические виртуальные/абстрактные члены интерфейсов (появившиеся в версии C# 11) обеспечивают *статический полиморфизм* — расширенное функциональное средство, которое обсуждается в главе 4. Статические абстрактные или виртуальные члены интерфейсов помечаются с помощью `static abstract` или `static virtual`:

```
interface ITypeDescribable
{
    static abstract string Description { get; }
    static virtual string Category => null;
}
```

В реализующем классе или структуре должны быть реализованы статические абстрактные члены и при необходимости могут быть реализованы статические виртуальные члены:

```
class CustomerTest : ITypeDescribable
{
    public static string Description => "Customer tests";      // Обязательно
    public static string Category     => "Unit testing";        // Необязательно
}
```

Помимо методов, свойств и событий, операции и преобразования также являются допустимыми целями для статических виртуальных членов интерфейсов (см. раздел “Перегрузка операций” главы 4). Статические виртуальные члены интерфейсов вызываются через ограниченный параметр типа, что будет демонстрироваться в разделах “Статический полиморфизм” и “Обобщенная математика” главы 4 после рассмотрения обобщений далее в текущей главе.

Написание кода класса или кода интерфейса

Запомните в качестве руководства следующие правила.

- Применяйте классы и подклассы для типов, которые естественным образом совместно используют некоторую реализацию.
- Применяйте интерфейсы для типов, которые имеют независимые реализации.

Рассмотрим показанные далее классы:

```
abstract class Animal {}
abstract class Bird           : Animal {}
abstract class Insect         : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore     : Animal {}
```

```
// Конкретные классы:  
class Ostrich : Bird {}  
class Eagle : Bird, FlyingCreature, Carnivore {} // Не допускается  
class Bee : Insect, FlyingCreature {} // Не допускается  
class Flea : Insect, Carnivore {} // Не допускается
```

Код классов `Eagle`, `Bee` и `Flea` не скомпилируется, потому что наследование от множества классов запрещено. Чтобы решить такую проблему, понадобится преобразовать некоторые типы в интерфейсы. Здесь и возникает вопрос: какие именно типы? Следуя главному правилу, мы можем сказать, что насекомые (`Insect`) разделяют реализацию и птицы (`Bird`) разделяют реализацию, а потому они остаются классами. В противоположность им летающие существа (`FlyingCreature`) имеют независимые механизмы для полета, а плотоядные животные (`Carnivore`) поддерживают независимые линии поведения при поедании, так что мы можем преобразовать `FlyingCreature` и `Carnivore` в интерфейсы:

```
interface IFlyingCreature {}  
interface ICarnivore {}
```

В типичном сценарии классы `Bird` и `Insect` могут соответствовать элементу управления Windows и веб-элементу управления, а `FlyingCreature` и `Carnivore` — интерфейсам `IPrintable` и `IUndoable`.

Перечисления

Перечисление — это специальный тип значения, который позволяет указывать группу именованных числовых констант, например:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Такое перечисление можно использовать следующим образом:

```
BorderSide topSide = BorderSide.Top;  
bool isTop = (topSide == BorderSide.Top); // true
```

Каждый член перечисления имеет лежащее в его основе целочисленное значение. По умолчанию:

- лежащие в основе значения относятся к типу `int`;
- членам перечисления присваиваются константы 0, 1, 2... (в порядке их объявления).

Можно указывать другой целочисленный тип:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

Для каждого члена перечисления можно явно указывать лежащие в основе значения:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```



Компилятор также позволяет явно присваивать значения *определенным* членам перечисления. Члены, которым не были присвоены значения, получают значения на основе инкрементирования последнего явно указанного значения. Предыдущий пример эквивалентен следующему коду:

```
public enum BorderSide : byte
{ Left=1, Right, Top=10, Bottom }
```

Преобразования перечислений

С помощью явного приведения экземпляр перечисления может быть преобразован в лежащее в основе целочисленное значение и из него:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

Можно также явно приводить один тип перечисления к другому. Предположим, что определение `HorizontalAlignment` выглядит следующим образом:

```
public enum HorizontalAlignment
{
    Left = BorderSide.Left,
    Right = BorderSide.Right,
    Center
}
```

При трансляции между типами перечислений используются лежащие в их основе целочисленные значения:

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// То же самое, что и:
HorizontalAlignment h = (HorizontalAlignment) (int) BorderSide.Right;
```

Числовой литерал 0 в выражении enum трактуется компилятором особым образом и явного приведения не требует:

```
BorderSide b = 0; // Приведение не требуется
if (b == 0) ...
```

Существуют две причины для специальной трактовки значения 0:

- первый член перечисления часто применяется как “стандартное” значение;
- для типов *комбинированных перечислений* значение 0 означает “отсутствие флагов”.

Перечисления флагов

Члены перечислений можно комбинировать. Чтобы предотвратить неоднозначности, члены комбинируемого перечисления требуют явного присваивания значений, обычно являющихся степенью двойки. Например:

```
[Flags]
public enum BorderSides { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

или:

```
enum BorderSides { None=0, Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3 }
```

При работе со значениями комбинированного перечисления используются побитовые операции, такие как | и &. Они имеют дело с лежащими в основе целыми значениями:

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;
if ((leftRight & BorderSides.Left) != 0)
    Console.WriteLine ("Includes Left");           // Включает Left
string formatted = leftRight.ToString();           // "Left, Right"
BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight);                // True
s ^= BorderSides.Right;                          // Переключает BorderSides.Right
Console.WriteLine (s);                           // Left
```

По соглашению к типу перечисления всегда должен применяться атрибут Flags, когда члены перечисления являются комбинируемыми. Если объявить такое перечисление без атрибута Flags, то комбинировать его члены по-прежнему можно будет, но вызов ToString на экземпляре перечисления приведет к выдаче числа, а не последовательности имен.

По соглашению типу комбинируемого перечисления назначается имя во множественном, а не единственном числе. Для удобства члены комбинаций могут быть помещены в само объявление перечисления:

```
[Flags]
enum BorderSides
{
    None=0,
    Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3,
    LeftRight = Left      | Right,
    TopBottom = Top       | Bottom,
    All        = LeftRight | TopBottom
}
```

Операции над перечислениями

Ниже указаны операции, которые могут работать с перечислениями:

```
=  ==  !=  <   >  <=  >=  +  -  ^  &  |  ~
+=  -=  +=  --  sizeof
```

Побитовые, арифметические и операции сравнения возвращают результат обработки лежащих в основе целочисленных значений. Сложение разрешено для перечисления и целочисленного типа, но не для двух перечислений.

Проблемы безопасности типов

Рассмотрим следующее перечисление:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Поскольку тип перечисления может быть приведен к лежащему в основе целочисленному типу и наоборот, фактическое значение может выходить за пределы допустимых границ законного члена перечисления:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b); // 12345
```

Побитовые и арифметические операции могут аналогично давать в результате недопустимые значения:

```
BorderSide b = BorderSide.Bottom;
b++; // Ошибки не возникают
```

Недопустимый экземпляр BorderSide может нарушить работу следующего кода:

```
void Draw (BorderSide side)
{
    if      (side == BorderSide.Left) {...}
    else if (side == BorderSide.Right){...}
    else if (side == BorderSide.Top)  {...}
    else                      {...} // Предполагается BorderSide.Bottom
}
```

Одно из решений предусматривает добавление дополнительной конструкции else:

```
...
else if (side == BorderSide.Bottom) ...
else throw new ArgumentException ("Invalid BorderSide: " + side, "side");
// Недопустимое значение BorderSide
```

Еще один обходной прием заключается в явной проверке значения перечисления на предмет допустимости. Такую работу выполняет статический метод Enum.IsDefined:

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side)); // False
```

К сожалению, метод Enum.IsDefined не работает с перечислениями флагов. Однако показанный далее вспомогательный метод (трюк, опирающийся на поведение Enum.ToString) возвращает true, если заданное перечисление флагов является допустимым:

```
for (int i = 0; i <= 16; i++)
{
    BorderSides side = (BorderSides)i;
    Console.WriteLine (IsFlagDefined (side) + " " + side);
}

bool IsFlagDefined (Enum e)
{
    decimal d;
    return !decimal.TryParse (e.ToString (), out d);
}

[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
```

Вложенные типы

Вложенный тип объявляется внутри области видимости другого типа:

```
public class TopLevel
{
    public class Nested { } // Вложенный класс
    public enum Color { Red, Blue, Tan } // Вложенное перечисление
}
```

Вложенный тип обладает следующими характеристиками.

- Он может получать доступ к закрытым членам включающего типа и ко всему остальному, к чему включающий тип имеет доступ.
- Он может быть объявлен с полным диапазоном модификаторов доступа, а не только `public` и `internal`.
- Стандартной доступностью вложенного типа является `private`, а не `internal`.
- Доступ к вложенному типу извне требует указания имени включающего типа (как при обращении к статическим членам).

Например, для доступа к члену `Color.Red` извне класса `TopLevel` необходимо записать такой код:

```
TopLevel.Color color = TopLevel.Color.Red;
```

Вложение в класс или структуру допускают все типы (классы, структуры, интерфейсы, делегаты и перечисления).

Ниже приведен пример обращения к закрытому члену типа из вложенного типа:

```
public class TopLevel
{
    static int x;
    class Nested
    {
        static void Foo() { Console.WriteLine (TopLevel.x); }
    }
}
```

А вот пример использования модификатора доступа `protected` с вложенным типом:

```
public class TopLevel
{
    protected class Nested { }

    public class SubTopLevel : TopLevel
    {
        static void Foo() { new TopLevel.Nested(); }
    }
}
```

Далее показан пример ссылки на вложенный тип извне включающего типа:

```
public class TopLevel
{
    public class Nested { }
}

class Test
{
    TopLevel.Nested n;
}
```

Вложенные типы интенсивно применяются самим компилятором, когда он генерирует закрытые классы, которые хранят состояние для таких конструкций, как итераторы и анонимные методы.



Если единственной причиной для использования вложенного типа является желание избежать загромождения пространства имен слишком большим числом типов, тогда взамен рассмотрите возможность применения вложенного пространства имен. Вложенный тип должен использоваться из-за его более строгих ограничений контроля доступа или же когда вложенному классу нужен доступ к закрытым членам включающего класса.

Обобщения

В C# имеются два отдельных механизма для написания кода, многократно применяемого различными типами: *наследование* и *обобщения*. В то время как наследование выражает повторное использование с помощью базового типа, обобщения делают это посредством “шаблона”, который содержит “типы-заполнители”. В сравнении с наследованием обобщения могут *увеличивать безопасность типов*, а также *сокращать количество приведений и упаковок*.



Обобщения C# и шаблоны C++ — похожие концепции, но работают они по-разному. Разница объясняется в разделе “Сравнение обобщений C# и шаблонов C++” в конце настоящей главы.

Обобщенные типы

Обобщенный тип объявляет *параметры типа* — типы-заполнители, предназначенные для заполнения потребителем обобщенного типа, который предоставляет *аргументы типа*. Ниже показан обобщенный тип `Stack<T>`, предназначенный для реализации стека экземпляров типа T. В `Stack<T>` объявлен единственный параметр типа T:

```
public class Stack<T>
{
    int position;
    T[] data = new T[100];
    public void Push (T obj) => data[position++] = obj;
    public T Pop()           => data[--position];
}
```

Вот как можно применять Stack<T>:

```
var stack = new Stack<int>();
stack.Push (5);
stack.Push (10);
int x = stack.Pop();      // x имеет значение 10
int y = stack.Pop();      // y имеет значение 5
```

Класс Stack<int> заполняет параметр типа T аргументом типа int, неявно создавая тип на лету (синтез происходит во время выполнения). Однако попытка помещения в стек типа Stack<int> строки приведет к ошибке на этапе компиляции. Фактически Stack<int> имеет показанное ниже определение (подстановки выделены полужирным, и во избежание путаницы вместо имени класса указано ###):

```
public class ###
{
    int position;
    int[] data;
    public void Push (int obj) => data[position++] = obj;
    public int Pop()           => data[--position];
}
```

Формально мы говорим, что Stack<T> — это *открытый (open) тип*, а Stack<int> — *закрытый (closed) тип*. Во время выполнения все экземпляры обобщенных типов закрываются — с заполнением их типов-заполнителей. Это значит, что показанный ниже оператор является недопустимым:

```
var stack = new Stack<T>(); // Не допускается: что собой представляет T?
```

Тем не менее, поступать так разрешено внутри класса или метода, который сам определяет T как параметр типа:

```
public class Stack<T>
{
    ...
    public Stack<T> Clone()
    {
        Stack<T> clone = new Stack<T>(); // Разрешено
        ...
    }
}
```

Для чего предназначены обобщения

Обобщения предназначены для написания кода, который может многократно использоваться различными типами. Предположим, что нам нужен стек целочисленных значений, но мы не располагаем обобщенными типами. Одно из решений предусматривает жесткое кодирование отдельной версии класса для каждого требуемого типа элементов (например, IntStack, StringStack и т.д.). Очевидно, что такой подход приведет к дублированию значительного объема кода. Другое решение заключается в написании стека, который обобщается за счет применения object в качестве типа элементов:

```
public class ObjectStack
{
    int position;
    object[] data = new object[10];
    public void Push (object obj) => data[position++] = obj;
    public object Pop()           => data[--position];
}
```

Тем не менее, класс ObjectStack не будет работать настолько же эффективно, как жестко закодированный класс IntStack, предназначенный для сохранения в стеке целочисленных значений. В частности, ObjectStack будет требовать упаковки и приведения вниз, которые не могут быть проверены на этапе компиляции:

```
// Предположим, что мы просто хотим сохранять целочисленные значения:
ObjectStack stack = new ObjectStack();

stack.Push ("s");           // Некорректный тип, но ошибка не возникает!
int i = (int)stack.Pop();   // Приведение вниз - ошибка времени выполнения
```

Нас интересует универсальная реализация стека, работающая со всеми типами элементов, а также возможность ее легкой специализации для конкретного типа элементов в целях повышения безопасности типов и сокращения приведений и упаковок. Именно это обеспечивают обобщения, позволяя параметризовать тип элементов. Тип Stack<T> обладает преимуществами и ObjectStack, и IntStack. Подобно ObjectStack класс Stack<T> написан один раз для универсальной работы со всеми типами. Как и IntStack, класс Stack<T> специализируется для конкретного типа — его элегантность заключается в том, что таким типом является T, который можно подставлять на лету.



Класс ObjectStack функционально эквивалентен Stack<object>.

Обобщенные методы

Обобщенный метод объявляет параметры типа внутри сигнатуры метода.

С помощью обобщенных методов многие фундаментальные алгоритмы могут быть реализованы единственным универсальным способом. Ниже показан обобщенный метод, который меняет местами содержимое двух переменных любого типа T:

```
static void Swap<T> (ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Метод Swap<T> можно вызывать следующим образом:

```
int x = 5;
int y = 10;
Swap (ref x, ref y);
```

Как правило, предоставлять аргументы типа обобщенному методу нет нужны, поскольку компилятор способен неявно вывести тип. Если имеется неоднозначность, то обобщенные методы могут быть вызваны с аргументами типа:

```
Swap<int> (ref x, ref y);
```

Внутри обобщенного *типа* метод не классифицируется как обобщенный, если только он не *вводит* параметры типа (посредством синтаксиса с угловыми скобками). Метод `Pop` в нашем обобщенном стеке просто задействует существующий параметр типа `T` и не трактуется как обобщенный.

Методы и типы — единственные конструкции, в которых могут вводиться параметры типа. Свойства, индексаторы, события, поля, конструкторы, операции и т.д. не могут объявлять параметры типа, хотя способны пользоваться любыми параметрами типа, которые уже объявлены во включающем типе. В примере с обобщенным стеком можно было бы написать индексатор, который возвращает обобщенный элемент:

```
public T this [int index] => data [index];
```

Аналогично конструкторы также могут пользоваться существующими параметрами типа, но не *вводить* их:

```
public Stack<T>() { } // Не допускается
```

Объявление параметров типа

Параметры типа могут вводиться в объявлениях классов, структур, интерфейсов, делегатов (рассматриваются в главе 4) и методов. Другие конструкции, такие как свойства, не могут *вводить* параметры типа, но могут их *использовать*. Например, свойство `Value` использует `T`:

```
public struct Nullable<T>
{
    public T Value { get; }
}
```

Обобщенный тип или метод может иметь несколько параметров:

```
class Dictionary< TKey, TValue > { ... }
```

Вот как создать его экземпляр:

```
Dictionary<int, string> myDict = new Dictionary<int, string>();
```

Или:

```
var myDict = new Dictionary<int, string>();
```

Имена обобщенных типов и методов могут быть перегружены при условии, что количество параметров типа у них отличается. Например, показанные ниже три имени типа не конфликтуют друг с другом:

```
class A      {}
class A<T>   {}
class A<T1, T2> {}
```