
Содержание

Об авторе	22
О научном редакторе	22
Благодарности	23
Предисловие	25
Структура книги	27
Книга для всех программирующих	27
Необходимое программное обеспечение	27
Код, доступный в Интернете	27
Что еще почитать	28
От издательства	28
ЧАСТЬ I. ЯЗЫК C#	29
Глава 1. Создание C#	31
Генеалогическое дерево C#	32
Язык C — начало современной эпохи программирования	32
Появление ООП и C++	33
Появление Интернета и Java	33
Создание C#	34
Развитие C#	36
Связь C# со средой .NET Framework	37
О среде NET Framework	37
Принцип действия CLR	38
Управляемый и неуправляемый код	38
Общезыковая спецификация	39
Глава 2. Краткий обзор элементов C#	41
Объектно-ориентированное программирование	41
Инкапсуляция	42
Полиморфизм	43
Наследование	44
Первая простая программа	44
Применение компилятора командной строки csc.exe	45
Применение интегрированной среды разработки Visual Studio	46
Построчный анализ первого примера программы	50
Обработка синтаксических ошибок	53
Незначительное изменение программы	54
Вторая простая программа	54

8 Содержание

Другие типы данных	57
Два управляющих оператора	58
Условный оператор	58
Оператор цикла	60
Использование кодовых блоков	61
Точка с запятой и оформление исходного текста программы	63
Ключевые слова C#	64
Идентификаторы	65
Библиотека классов среды .NET Framework	66
Глава 3. Типы данных, литералы и переменные	67
О значении типов данных	67
Типы значений в C#	68
Целочисленные типы	69
Типы для представления чисел с плавающей точкой	71
Десятичный тип данных	73
Символы	74
Логический тип данных	75
Некоторые возможности вывода	76
Литералы	79
Шестнадцатеричные литералы	80
Управляющие последовательности символов	80
Строковые литералы	81
Более подробное рассмотрение переменных	83
Инициализация переменной	83
Динамическая инициализация	84
Неявно типизированные переменные	85
Область действия и время существования переменных	86
Преобразование и приведение типов	89
Автоматическое преобразование типов	90
Приведение несовместимых типов	91
Преобразование типов в выражениях	93
Приведение типов в выражениях	95
Глава 4. Операторы	97
Арифметические операторы	97
Операторы инкремента и декремента	98
Операторы отношения и логические операторы	101
Укороченные логические операторы	104
Оператор присваивания	106
Составные операторы присваивания	107
Поразрядные операторы	107
Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ	108
Операторы сдвига	114

Поразрядные составные операторы присваивания	117
Оператор ?	117
Использование пробелов и круглых скобок	119
Предшествование операторов	119
Глава 5. Управляющие операторы	121
Оператор if	121
Вложенные операторы if	122
Конструкция if-else-if	124
Оператор switch	125
Вложенные операторы switch	129
Оператор цикла for	129
Некоторые разновидности оператора цикла for	131
Оператор цикла while	137
Оператор цикла do-while	138
Оператор цикла foreach	139
Применение оператора break для выхода из цикла	139
Применение оператора continue	142
Оператор return	143
Оператор goto	143
Глава 6. Введение в классы, объекты и методы	147
Основные положения о классах	147
Общая форма определения класса	148
Определение класса	149
Создание объектов	153
Переменные ссылочного типа и присваивание	154
Методы	155
Добавление метода в класс Building	156
Возврат из метода	158
Возврат значения	159
Использование параметров	162
Добавление параметризованного метода в класс Building	164
Исключение недоступного кода	166
Конструкторы	166
Параметризованные конструкторы	168
Добавление конструктора в класс Building	169
Еще раз об операторе new	170
Применение оператора new вместе с типами значений	170
“Сборка мусора” и применение деструкторов	171
Деструкторы	172
Ключевое слово this	174
Глава 7. Массивы и строки	177
Массивы	177

Одномерные массивы	178
Многомерные массивы	182
Двумерные массивы	182
Массивы трех и более измерений	183
Инициализация многомерных массивов	184
Ступенчатые массивы	185
Присваивание ссылок на массивы	187
Применение свойства Length	189
Применение свойства Length при обращении со ступенчатыми массивами	191
Неявно типизированные массивы	192
Оператор цикла foreach	194
Строки	198
Построение строк	198
Обращение со строками	199
Массивы строк	203
Постоянство строк	205
Применение строк в операторах switch	206
Глава 8. Подробнее о методах и классах	209
Управление доступом к членам класса	209
Модификаторы доступа	210
Организация закрытого и открытого доступа	212
Практический пример организации управления доступом	212
Передача объектов методам по ссылке	218
Способы передачи аргументов методу	220
Использование модификаторов параметров ref и out	222
Использование модификатора параметра ref	223
Использование модификатора параметра out	224
Использование модификаторов ref и out для ссылок на объекты	227
Использование переменного числа аргументов	229
Возврат объектов из методов	231
Возврат массива из метода	234
Перегрузка методов	235
Перегрузка конструкторов	241
Вызов перегружаемого конструктора с помощью ключевого слова this	245
Инициализаторы объектов	246
Необязательные аргументы	247
Необязательные аргументы и перегрузка методов	249
Необязательные аргументы и неоднозначность	250
Практический пример использования необязательных аргументов	251
Именованные аргументы	252
Метод Main()	254
Возврат значений из метода Main()	254

Передача аргументов методу Main()	255
Рекурсия	257
Применение ключевого слова static	260
Статические конструкторы	265
Статические классы	266
Глава 9. Перегрузка операторов	269
Основы перегрузки операторов	270
Перегрузка бинарных операторов	270
Перегрузка унарных операторов	273
Выполнение операций со встроенными в C# типами данных	277
Перегрузка операторов отношения	281
Перегрузка операторов true и false	283
Перегрузка логических операторов	286
Простой способ перегрузки логических операторов	286
Как сделать укороченные логические операторы доступными для применения	288
Операторы преобразования	292
Рекомендации и ограничения по перегрузке операторов	297
Еще один пример перегрузки операторов	298
Глава 10. Индексаторы и свойства	303
Индексаторы	303
Создание одномерных индексаторов	304
Перегрузка индексаторов	307
Индексаторы без базового массива	310
Многомерные индексаторы	311
Свойства	313
Автоматически реализуемые свойства	318
Применение инициализаторов объектов в свойствах	319
Ограничения, присущие свойствам	320
Применение модификаторов доступа в аксессуарах	320
Применение индексаторов и свойств	324
Глава 11. Наследование	329
Основы наследования	329
Доступ к членам класса и наследование	333
Организация защищенного доступа	336
Конструкторы и наследование	337
Вызов конструкторов базового класса	339
Наследование и сокрытие имен	343
Применение ключевого слова base для доступа к скрытому имени	344
Создание многоуровневой иерархии классов	346
Порядок вызова конструкторов	349
Ссылки на базовый класс и объекты производных классов	351

Виртуальные методы и их переопределение	355
Что дает переопределение методов	359
Применение виртуальных методов	360
Применение абстрактных классов	363
Предотвращение наследования с помощью ключевого слова sealed	367
Класс object	368
Упаковка и распаковка	370
Класс object как универсальный тип данных	372
Глава 12. Интерфейсы, структуры и перечисления	375
Интерфейсы	375
Реализация интерфейсов	377
Применение интерфейсных ссылок	381
Интерфейсные свойства	383
Интерфейсные индексаторы	385
Наследование интерфейсов	387
Соккрытие имен при наследовании интерфейсов	388
Явные реализации	388
Выбор между интерфейсом и абстрактным классом	391
Стандартные интерфейсы для среды .NET Framework	391
Структуры	391
О назначении структур	395
Перечисления	397
Инициализация перечисления	399
Указание базового типа перечисления	399
Применение перечислений	399
Глава 13. Обработка исключительных ситуаций	403
Класс System.Exception	404
Основы обработки исключительных ситуаций	404
Применение пары ключевых слов try и catch	404
Простой пример обработки исключительной ситуации	405
Второй пример обработки исключительной ситуации	407
Последствия перехвата исключений	408
Обработка исключительных ситуаций — “изящный” способ устранения программных ошибок	410
Применение нескольких операторов catch	411
Перехват всех исключений	412
Вложение блоков try	413
Генерирование исключений вручную	414
Повторное генерирование исключений	415
Использование блока finally	416
Подробное рассмотрение класса Exception	418
Наиболее часто используемые исключения	420

Получение производных классов исключений	422
Перехват исключений производных классов	426
Применение ключевых слов checked и unchecked	428
Глава 14. Применение средств ввода-вывода	431
Организация системы ввода-вывода в C# на потоках	431
Байтовые и символьные потоки	432
Встроенные потоки	432
Классы потоков	432
Класс Stream	432
Классы байтовых потоков	434
Классы-оболочки символьных потоков	434
Двоичные потоки	436
Консольный ввод-вывод	436
Чтение данных из потока ввода с консоли	436
Применение метода ReadKey()	438
Запись данных в поток вывода на консоль	440
Класс FileStream и байтовый ввод-вывод в файл	441
Открытие и закрытие файла	441
Чтение байтов из потока файлового ввода-вывода	444
Запись в файл	446
Использование класса FileStream для копирования файла	448
Символьный ввод-вывод в файл	449
Применение класса StreamWriter	449
Применение класса StreamReader	451
Переадресация стандартных потоков	452
Чтение и запись двоичных данных	454
Класс BinaryWriter	454
Класс BinaryReader	455
Демонстрирование двоичного ввода-вывода	457
Файлы с произвольным доступом	461
Применение класса MemoryStream	463
Применение классов StringReader и StringWriter	465
Класс File	467
Копирование файлов с помощью метода Copy()	467
Применение методов Exists() и GetLastAccessTime()	468
Преобразование числовых строк в их внутреннее представление	469
Глава 15. Делегаты, события и лямбда-выражения	473
Делегаты	473
Групповое преобразование делегируемых методов	476
Применение методов экземпляра в качестве делегатов	477
Групповая адресация	478
Ковариантность и контравариантность	481

Класс System.Delegate	483
Назначение делегатов	483
Анонимные функции	483
Анонимные методы	484
Передача аргументов анонимному методу	484
Возврат значения из анонимного метода	485
Применение внешних переменных в анонимных методах	486
Лямбда-выражения	488
Лямбда-оператор	488
Одиночные лямбда-выражения	489
Блочные лямбда-выражения	492
События	494
Пример групповой адресации события	496
Методы экземпляра в сравнении со статическими методами в качестве обработчиков событий	497
Применение аксессуаров событий	500
Разнообразные возможности событий	504
Применение анонимных методов и лямбда-выражений вместе с событиями	504
Рекомендации по обработке событий в среде .NET Framework	506
Применение делегатов EventHandler<EventArgs> и EventHandler	508
Практический пример обработки событий	509
Глава 16. Пространства имен, препроцессор и сборки	513
Пространства имен	513
Объявление пространства имен	514
Предотвращение конфликтов имен с помощью пространств имен	516
Директива using	518
Вторая форма директивы using	520
Аддитивный характер пространств имен	521
Вложенные пространства имен	523
Глобальное пространство имен	524
Применение описателя псевдонима пространства имен ::	524
Препроцессор	528
Директива #define	529
Директивы #if и #endif	529
Директивы #else и #elif	531
Директива #undef	533
Директива #error	533
Директива #warning	534
Директива #line	534
Директивы #region и #endregion	534
Директива #pragma	534
Сборки и модификатор доступа internal	535

Модификатор доступа <code>internal</code>	536
Глава 17. Динамическая идентификация типов, рефлексия и атрибуты	537
Динамическая идентификация типов	537
Проверка типа с помощью оператора <code>is</code>	538
Применение оператора <code>as</code>	539
Применение оператора <code>typeof</code>	540
Рефлексия	541
Класс <code>System.Type</code> — ядро подсистемы рефлексии	541
Применение рефлексии	543
Получение сведений о методах	544
Вызов методов с помощью рефлексии	548
Получение конструкторов конкретного типа	550
Получение типов данных из сборок	555
Полностью автоматизированное обнаружение типов	560
Атрибуты	562
Основы применения атрибутов	563
Сравнение позиционных и именованных параметров	566
Встроенные атрибуты	570
Атрибут <code>AttributeUsage</code>	570
Атрибут <code>Conditional</code>	571
Атрибут <code>Obsolete</code>	572
Глава 18. Обобщения	575
Что такое обобщения	576
Простой пример обобщений	576
Различение обобщенных типов по аргументам типа	580
Повышение типовой безопасности с помощью обобщений	580
Обобщенный класс с двумя параметрами типа	583
Общая форма обобщенного класса	585
Ограниченные типы	585
Применение ограничения на базовый класс	586
Применение ограничения на интерфейс	594
Применение ограничения <code>new()</code> на конструктор	598
Ограничения ссылочного типа и типа значения	599
Установление связи между двумя параметрами типа с помощью ограничения	602
Применение нескольких ограничений	603
Получение значения, присваиваемого параметру типа по умолчанию	604
Обобщенные структуры	606
Создание обобщенного метода	607
Вызов обобщенного метода с явно указанными аргументами типа	609
Применение ограничений в обобщенных методах	610
Обобщенные делегаты	610

Обобщенные интерфейсы	612
Сравнение экземпляров параметра типа	615
Иерархии обобщенных классов	620
Применение обобщенного базового класса	620
Обобщенный производный класс	622
Переопределение виртуальных методов в обобщенном классе	623
Перегрузка методов с несколькими параметрами типа	625
Ковариантность и контравариантность в параметрах обобщенного типа	626
Применение ковариантности в обобщенном интерфейсе	626
Применение контравариантности в обобщенном интерфейсе	630
Вариантные делегаты	633
Создание экземпляров объектов обобщенных типов	635
Некоторые ограничения, присущие обобщениям	636
Заключительные соображения относительно обобщений	636
Глава 19. LINQ	637
Основы LINQ	638
Простой запрос	639
Неоднократное выполнение запросов	641
Связь между типами данных в запросе	642
Общая форма запроса	643
Отбор запрашиваемых значений с помощью оператора where	644
Сортировка результатов запроса с помощью оператора orderby	646
Подробное рассмотрение оператора select	649
Применение вложенных операторов from	653
Группирование результатов с помощью оператора group	655
Продолжение запроса с помощью оператора into	657
Применение оператора let для создания временной переменной в запросе	659
Объединение двух последовательностей с помощью оператора join	660
Анонимные типы	663
Создание группового объединения	666
Методы запроса	669
Основные методы запроса	669
Формирование запросов с помощью методов запроса	670
Синтаксис запросов и методы запроса	673
Дополнительные методы расширения, связанные с запросами	673
Режимы выполнения запросов: отложенный и немедленный	675
Деревья выражений	676
Методы расширения	678
PLINQ	680

Глава 20. Небезопасный код, указатели, обнуляемые типы и разные ключевые слова	681
Небезопасный код	681
Основы применения указателей	682
Применение ключевого слова unsafe	684
Применение модификатора fixed	685
Доступ к членам структуры с помощью указателя	686
Арифметические операции над указателями	686
Сравнение указателей	688
Указатели и массивы	688
Указатели и строки	690
Многоуровневая непрямая адресация	691
Массивы указателей	692
Создание буферов фиксированного размера	693
Обнуляемые типы	695
Основы применения обнуляемых типов	695
Применение обнуляемых объектов в выражениях	697
Оператор ??	698
Обнуляемые объекты, операторы отношения и логические операторы	699
Частичные типы	700
Частичные методы	701
Создание объектов динамического типа	703
Возможность взаимодействия с моделью COM	707
Дружественные сборки	708
Разные ключевые слова	708
Ключевое слово lock	708
Ключевое слово readonly	709
Ключевые слова const и volatile	710
Оператор using	711
Ключевое слово extern	712
ЧАСТЬ II. БИБЛИОТЕКА C#	717
Глава 21. Пространство имен System	719
Члены пространства имен System	720
Класс Math	721
Структуры .NET, соответствующие встроенным типам значений	727
Структуры целочисленных типов данных	728
Структуры типов данных с плавающей точкой	730
Структура Decimal	735
Структура Char	741
Структура Boolean	748
Класс Array	750
Сортировка и поиск в массивах	763

Обращение содержимого массива	766
Копирование массива	767
Применение предиката	768
Применение делегата Action	769
Класс BitConverter	771
Генерирование случайных чисел средствами класса Random	773
Управление памятью и класс GC	774
Класс object	776
Класс Tuple	777
Интерфейсы IComparable и IComparable<T>	778
Интерфейс IEquatable<T>	778
Интерфейс IConvertible	779
Интерфейс ICloneable	779
Интерфейсы IFormatProvider и IFormattable	781
Интерфейсы IObservable<T> и IObservable<T>	781
Глава 22. Строки и форматирование	783
Строки в C#	783
Класс String	784
Конструкторы класса String	784
Поле, индексатор и свойство класса String	785
Операторы класса String	786
Заполнение и обрезка строк	808
Вставка, удаление и замена строк	810
Смена регистра	811
Применение метода Substring()	811
Методы расширения класса String	812
Форматирование	812
Общее представление о форматировании	812
Спецификаторы формата числовых данных	814
Представление о номерах аргументов	815
Применение методов String.Format() и ToString() для форматирования данных	816
Применение метода String.Format() для форматирования значений	816
Применение метода ToString() для форматирования данных	819
Определение пользовательского формата числовых данных	820
Символы-заполнители специального формата числовых данных	820
Форматирование даты и времени	824
Определение пользовательского формата даты и времени	827
Форматирование промежутков времени	829
Форматирование перечислений	830
Глава 23. Многопоточное программирование. Часть первая: основы	833
Основы многопоточной обработки	834

Класс Thread	835
Создание и запуск потока	836
Простые способы усовершенствования многопоточной программы	838
Создание нескольких потоков	839
Определение момента окончания потока	841
Передача аргумента потоку	844
Свойство IsBackground	846
Приоритеты потоков	847
Синхронизация	849
Другой подход к синхронизации потоков	853
Класс Monitor и блокировка	855
Сообщение между потоками с помощью методов Wait(), Pulse() и PulseAll()	855
Пример использования методов Wait() и Pulse()	856
Взаимоблокировка и состояние гонки	860
Применение атрибутаMethodImplAttribute	860
Применение мьютекса и семафора	862
Мьютекс	863
Семафор	867
Применение событий	870
Класс Interlocked	873
Классы синхронизации, внедренные в версии .NET Framework 4.0	874
Прерывание потока	875
Другая форма метода Abort()	876
Отмена действия метода Abort()	878
Приостановка и возобновление потока	880
Определение состояния потока	880
Применение основного потока	880
Дополнительные средства многопоточной обработки, внедренные в версии .NET Framework 4.0	882
Рекомендации по многопоточному программированию	882
Запуск отдельной задачи	882
Глава 24. Многопоточное программирование. Часть вторая: библиотека TPL	885
Два подхода к параллельному программированию	886
Класс Task	887
Создание задачи	887
Применение идентификатора задачи	890
Применение методов ожидания	891
Вызов метода Dispose()	895
Применение класса TaskFactory для запуска задачи	895
Применение лямбда-выражения в качестве задачи	896
Создание продолжения задачи	897

Возврат значения из задачи	899
Отмена задачи и обработка исключения AggregateException	901
Другие средства организации задач	905
Класс Parallel	906
Распараллеливание задач методом Invoke()	906
Применение метода For()	909
Применение метода ForEach()	915
Исследование возможностей PLINQ	917
Класс ParallelEnumerable	917
Распараллеливание запроса методом AsParallel()	918
Применение метода AsOrdered()	919
Отмена параллельного запроса	920
Другие средства PLINQ	922
Вопросы эффективности PLINQ	922
Глава 25. Коллекции, перечислители и итераторы	923
Краткий обзор коллекций	924
Необобщенные коллекции	925
Интерфейсы необобщенных коллекций	926
Структура DictionaryEntry	931
Классы необобщенных коллекций	931
Хранение отдельных битов в классе коллекции BitArray	950
Специальные коллекции	953
Обобщенные коллекции	954
Интерфейсы обобщенных коллекций	954
Структура KeyValuePair<TKey, TValue>	960
Классы обобщенных коллекций	960
Параллельные коллекции	983
Сохранение объектов, определяемых пользователем классов, в коллекции	988
Реализация интерфейса IComparable	990
Реализация интерфейса IComparable для необобщенных коллекций	990
Реализация интерфейса IComparable для обобщенных коллекций	992
Применение интерфейса IComparer	994
Применение необобщенного интерфейса IComparer	994
Применение обобщенного интерфейса IComparer<T>	996
Применение класса StringComparer	997
Доступ к коллекции с помощью перечислителя	998
Применение обычного перечислителя	999
Применение перечислителя типа IDictionaryEnumerator	1000
Реализация интерфейсов IEnumerable и IEnumerator	1001
Применение итераторов	1003
Прерывание итератора	1005
Применение нескольких операторов yield	1006

Создание именованного итератора	1006
Создание обобщенного итератора	1008
Инициализаторы коллекций	1009
Глава 26. Сетевые средства подключения к Интернету	1011
Члены пространства имен System.Net	1012
Универсальные идентификаторы ресурсов	1013
Основы организации доступа к Интернету	1014
Класс WebRequest	1015
Класс WebResponse	1017
Классы HttpRequest и HttpResponse	1018
Первый простой пример	1018
Обработка сетевых ошибок	1021
Исключения, генерируемые методом Create()	1021
Исключения, генерируемые методом GetResponse()	1022
Исключения, генерируемые методом GetResponseStream()	1022
Обработка исключений	1022
Класс Uri	1024
Доступ к дополнительной информации, получаемой в ответ по протоколу HTTP	1025
Доступ к заголовку	1026
Доступ к cookie-наборам	1027
Применение свойства LastModified	1029
Практический пример создания программы MiniCrawler	1030
Применение класса WebClient	1034
Приложение. Краткий справочник по составлению документирующих комментариев	1039
Дескрипторы XML-комментариев	1039
Компилирование документирующих комментариев	1041
Пример составления документации в формате XML	1041
Предметный указатель	1044

Краткий обзор элементов С#

Наибольшие трудности в изучении языка программирования вызывает то обстоятельство, что ни один из его элементов не существует обособленно. Напротив, все элементы языка действуют совместно. Такая взаимосвязанность затрудняет рассмотрение одного аспекта С# безотносительно к другому. Поэтому для преодоления данного затруднения в этой главе дается краткий обзор нескольких средств языка С#, включая общую форму программы на С#, ряд основных управляющих и прочих операторов. Вместо того чтобы углубляться в детали, в этой главе основное внимание уделяется лишь самым общим принципам написания любой программы на С#. А большинство вопросов, затрагиваемых по ходу изложения материала этой главы, более подробно рассматриваются в остальных главах части I.

Объектно-ориентированное программирование

Основным понятием С# является объектно-ориентированное программирование (ООП). Методика ООП неотделима от С#, и поэтому все программы на С# являются объектно-ориентированными хотя бы в самой малой степени. В связи с этим очень важно и полезно усвоить основополагающие принципы ООП, прежде чем приступать к написанию самой простой программы на С#.

ООП представляет собой эффективный подход к программированию. Методики программирования претерпели существенные изменения с момента изобретения

компьютера, постепенно приспособляясь, главным образом, к повышению сложности программ. Когда, например, появились первые ЭВМ, программирование заключалось в ручном переключении на разные двоичные машинные команды с переднего пульта управления ЭВМ. Такой подход был вполне оправданным, поскольку программы состояли всего из нескольких сотен команд. Дальнейшее усложнение программ привело к разработке языка ассемблера, который давал программистам возможность работать с более сложными программами, используя символическое представление отдельных машинных команд. Постоянное усложнение программ вызвало потребность в разработке и внедрении в практику программирования таких языков высокого уровня, как, например, FORTRAN и COBOL, которые предоставляли программистам больше средств для того, чтобы как-то справиться с постоянно растущей сложностью программ. Но как только возможности этих первых языков программирования были полностью исчерпаны, появились разработки языков структурного программирования, в том числе и C.

На каждом этапе развития программирования появлялись методы и инструментальные средства для “обуздания” растущей сложности программ. И на каждом таком этапе новый подход вбирал в себя все самое лучшее из предыдущих, знаменуя собой прогресс в программировании. Это же можно сказать и об ООП. До ООП многие проекты достигали (а иногда и превышали) предел, за которым структурный подход к программированию оказывался уже неработоспособным. Поэтому для преодоления трудностей, связанных с усложнением программ, и возникла потребность в ООП.

ООП вобрало в себя все самые лучшие идеи структурного программирования, объединив их с рядом новых понятий. В итоге появился новый и лучший способ организации программ. В самом общем виде программа может быть организована одним из двух способов: вокруг кода (т.е. того, что фактически происходит) или же вокруг данных (т.е. того, что подвергается воздействию). Программы, созданные только методами структурного программирования, как правило, организованы вокруг кода. Такой подход можно рассматривать “как код, воздействующий на данные”.

Совсем иначе работают объектно-ориентированные программы. Они организованы вокруг данных, исходя из главного принципа: “данные управляют доступом к коду”. В объектно-ориентированном языке программирования определяются данные и код, которому разрешается воздействовать на эти данные. Следовательно, тип данных точно определяет операции, которые могут быть выполнены над данными.

Для поддержки принципов ООП все объектно-ориентированные языки программирования, в том числе и C#, должны обладать тремя общими свойствами: инкапсуляцией, полиморфизмом и наследованием. Рассмотрим каждое из этих свойств в отдельности.

Инкапсуляция

Инкапсуляция — это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. В объектно-ориентированном языке данные и код могут быть объединены в совершенно автономный *черный ящик*. Внутри такого ящика находятся все необходимые данные и код. Когда код и данные связываются вместе подобным образом, создается объект. Иными словами, *объект* — это элемент, поддерживающий инкапсуляцию.

В объекте код, данные или же и то и другое могут быть *закрытыми* или же *открытыми*. Закрытые данные или код известны и доступны только остальной части объекта. Это означает, что закрытые данные или код недоступны части программы, находящейся за пределами объекта. Если же данные или код оказываются открытыми, то они доступны другим частям программы, хотя и определены внутри объекта. Как правило, открытые части объекта служат для организации управляемого интерфейса с закрытыми частями.

Основной единицей инкапсуляции в С# является *класс*, который определяет форму объекта. Он описывает данные, а также код, который будет ими оперировать. В С# описание класса служит для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта.

Код и данные, составляющие вместе класс, называют *членами*. Данные, определяемые классом, называют *полями*, или *переменными экземпляра*. А код, оперирующий данными, содержится в *функциях-членах*, самым типичным представителем которых является *метод*. В С# метод служит в качестве аналога подпрограммы. (К числу других функций-членов относятся свойства, события и конструкторы.) Таким образом, методы класса содержат код, воздействующий на поля, определяемые этим классом.

Полиморфизм

Полиморфизм, что по-гречески означает “множество форм”, — это свойство, позволяющее одному интерфейсу получать доступ к общему классу действий. Простым примером полиморфизма может служить руль автомашины, который выполняет одни и те же функции своеобразного интерфейса независимо от вида применяемого механизма управления автомашиной. Это означает, что руль действует одинаково независимо от вида рулевого управления: прямого действия, с усилением или реечной передачей. Следовательно, при вращении руля влево автомашина всегда поворачивает влево, какой бы вид управления в ней ни применялся. Главное преимущество единообразного интерфейса заключается в том, что, зная, как обращаться с рулем, вы сумеете водить автомашину любого типа.

Тот же самый принцип может быть применен и в программировании. Рассмотрим для примера *стек*, т.е. область памяти, функционирующую по принципу “последним пришел — первым обслужен”. Допустим, что в программе требуются три разных типа стеков: один — для целых значений, другой — для значений с плавающей точкой, третий — для символьных значений. В данном примере алгоритм, реализующий все эти стеки, остается неизменным, несмотря на то, что в них сохраняются разнотипные данные. В языке, не являющемся объектно-ориентированным, для этой цели пришлось бы создать три разных набора стековых подпрограмм с разными именами. Но благодаря полиморфизму для реализации всех трех типов стеков в С# достаточно создать лишь один общий набор подпрограмм. Зная, как пользоваться одним стеком, вы сумеете воспользоваться и остальными.

В более общем смысле понятие полиморфизма нередко выражается следующим образом: “один интерфейс — множество методов”. Это означает, что для группы взаимосвязанных действий можно разработать общий интерфейс. Полиморфизм помогает упростить программу, позволяя использовать один и тот же интерфейс для описания *общего класса действий*. Выбрать конкретное действие (т.е. метод) в каждом отдельном случае — это задача компилятора. Программисту не нужно делать это самому. Ему достаточно запомнить и правильно использовать общий интерфейс.

Наследование

Наследование представляет собой процесс, в ходе которого один объект приобретает свойства другого объекта. Это очень важный процесс, поскольку он обеспечивает принцип иерархической классификации. Если вдуматься, то большая часть знаний поддается систематизации благодаря иерархической классификации по нисходящей. Например, сорт яблок “Джонатан” входит в общую классификацию сортов яблок, которые, в свою очередь, относятся к классу *фруктов*, а те — к еще более крупному классу *пищевых продуктов*. Это означает, что класс пищевых продуктов обладает рядом свойств (съедобности, питательности и т.д.), которые по логике вещей распространяются и на его подкласс фруктов. Помимо этих свойств, класс фруктов обладает своими собственными свойствами (сочностью, сладостью и т.д.), которыми он отличается от других пищевых продуктов. У класса яблок имеются свои характерные особенности (растут на деревьях, не в тропиках и т.д.). Таким образом, сорт яблок “Джонатан” наследует свойства всех предшествующих классов, обладая в то же время свойствами, присущими только этому сорту яблок, например красной окраской кожицы с желтым бочком и характерным ароматом и вкусом.

Если не пользоваться иерархиями, то для каждого объекта пришлось бы явно определять все его свойства. А если воспользоваться наследованием, то достаточно определить лишь те свойства, которые делают объект особенным в его классе. Он может также наследовать общие свойства своего родителя. Следовательно, благодаря механизму наследования один объект становится отдельным экземпляром более общего класса.

Первая простая программа

А теперь самое время перейти к примеру конкретной программы на C#. Для начала скомпилируем и выполним короткую программу.

```
/*
  Это простая программа на C#.
  Назовем ее Example.cs.
*/

using System;

class Example {

    // Любая программа на C# начинается с вызова метода Main().
    static void Main() {
        Console.WriteLine("Простая программа на C#.");
    }
}
```

Основной средой для разработки программ на C# служит Visual Studio корпорации Microsoft. Для компилирования примеров всех программ, приведенных для примера в этой книге, в том числе и тех, где используются новые средства C# 4.0, вам потребуется Visual Studio 2010 или же более поздняя версия, поддерживающая C#.

Создавать, компилировать и выполнять программы на C#, используя Visual Studio, можно двумя способами: пользуясь, во-первых, интегрированной средой разработки Visual Studio, а во-вторых, — компилятором командной строки `csc.exe`. Далее описываются оба способа.

Применение компилятора командной строки `csc.exe`

Для коммерческой разработки программ вам, скорее всего, придется пользоваться интегрированной средой Visual Studio, хотя для некоторых читателей более удобным может оказаться компилятор, работающий в режиме командной строки, особенно для компилирования и выполнения примеров программ, приведенных в этой книге. Объясняется это тем, что для работы над отдельной программой не нужно создавать целый проект. Для этого достаточно написать программу, а затем скомпилировать и выполнить ее, причем все это делается из командной строки. Таким образом, если вы умеете пользоваться окном Командная строка (Command Prompt) и его интерфейсом в Windows, то компилятор командной строки окажется для вас более простым и оперативным инструментальным средством, чем интегрированная среда разработки.

ПРЕДОСТЕРЕЖЕНИЕ

Если вы не знаете, как пользоваться окном Командная строка, то вам лучше работать в интегрированной среде разработки Visual Studio. Ведь пытаться усвоить одновременно команды интерфейса Командная строка и элементы языка C# не так-то просто, несмотря на то, что запомнить эти команды совсем нетрудно.

Для написания и выполнения программ на C# с помощью компилятора командной строки выполните следующую несложную процедуру.

1. Введите исходный текст программы, используя текстовый редактор.
2. Скомпилируйте программу с помощью компилятора `csc.exe`.
3. Выполните программу.

Ввод исходного текста программы

Исходный текст примеров программ, приведенных в этой книге, доступен для загрузки по адресу www.mhprofessional.com. Но при желании вы можете сами ввести исходный текст этих программ вручную. Для этого воспользуйтесь избранным текстовым редактором, например Notepad. Но не забывайте, что вы должны создать файлы, содержащие простой, а не отформатированный текст, поскольку информация форматирования текста, сохраняемая в файле для обработки текста, может помешать нормальной работе компилятора C#. Введя исходный текст программы, присвойте ее файлу имя `Example.cs`.

Компилирование программы

Для компилирования программы на C# запустите на выполнение компилятор `csc.exe`, указав имя исходного файла в командной строке.

```
C:\>csc Example.cs
```

Компилятор `csc` создаст файл `Example.exe`, содержащий версию MSIL данной программы. Несмотря на то что псевдокод MSIL не является исполняемым кодом, он содержится в исполняемом файле с расширением `.exe`. Среда CLR автоматически вызывает JIT-компилятор при попытке выполнить файл `Example.exe`. Следует, однако, иметь в виду, что если попытаться выполнить файл `Example.exe` (или любой другой

исполняемый файл, содержащий псевдокод MSIL) на том компьютере, где среда .NET Framework не установлена, то программа не будет выполнена, поскольку на этом компьютере отсутствует среда CLR.

ПРИМЕЧАНИЕ

Прежде чем запускать на выполнение компилятор `csc.exe`, откройте окно Командная строка, настроенное под Visual Studio. Для этого проще всего выбрать команду Visual Studio ⇒ Инструменты Visual Studio ⇒ Командная строка Visual Studio (Visual Studio ⇒ Visual Studio Tools ⇒ Visual Studio Command Prompt) из меню Пуск ⇒ Все программы (Start ⇒ All Programs) на панели задач Windows. Кроме того, вы можете открыть ненастроенное окно Командная строка, а затем выполнить командный файл `vsvars32.bat`, входящий в состав Visual Studio.

Выполнение программы

Для выполнения программы введите ее имя в командной строке следующим образом.

```
C:\>Example
```

В результате выполнения программы на экране появится такая строка.

Простая программа на C#.

Применение интегрированной среды разработки Visual Studio

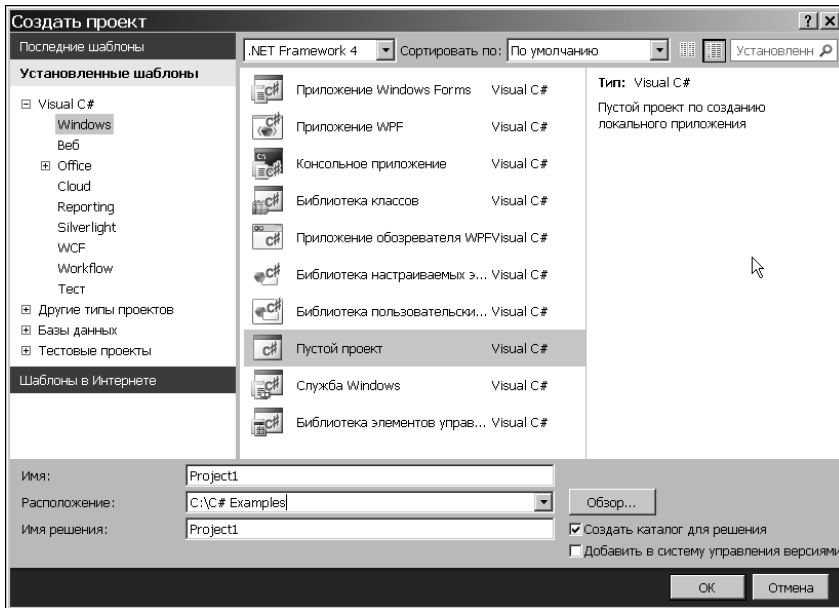
Visual Studio представляет собой интегрированную среду разработки программ, созданную корпорацией Microsoft. Такая среда дает возможность править, компилировать, выполнять и отлаживать программы на C#, не покидая эту грамотно организованную среду. Visual Studio предоставляет не только все необходимые средства для работы с программами, но и помогает правильно организовать их. Она оказывается наиболее эффективной для работы над крупными проектами, хотя может быть с тем же успехом использована и для разработки небольших программ, например, тех, которые приведены в качестве примера в этой книге.

Ниже приведена краткая процедура правки, компилирования и выполнения программы на C# в интегрированной среде разработки Visual Studio 2010. При этом предполагается, что интегрированная среда разработки входит в состав пакета Visual Studio 2010 Professional. В других версиях Visual Studio возможны незначительные отличия.

1. Создайте новый (пустой) проект C#, выбрав команду **Файл** ⇒ **Создать** ⇒ **Проект** (File ⇒ New ⇒ Project). Затем выберите элемент **Windows** из списка **Установленные шаблоны** (Installed Templates) и далее — шаблон **Пустой проект** (Empty Project), как показано на рисунке.

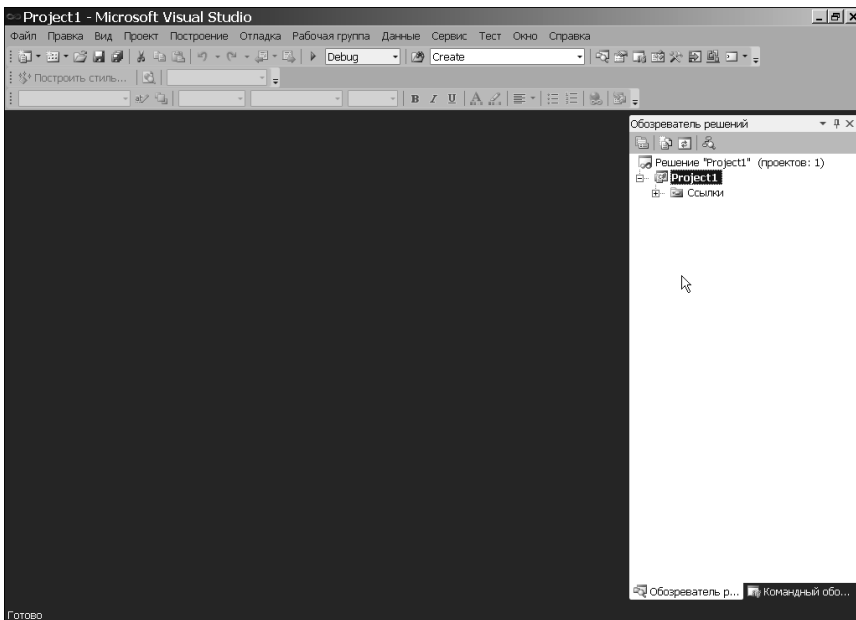
ПРИМЕЧАНИЕ

Имя и местоположение вашего проекта может отличаться от того, что показано здесь.



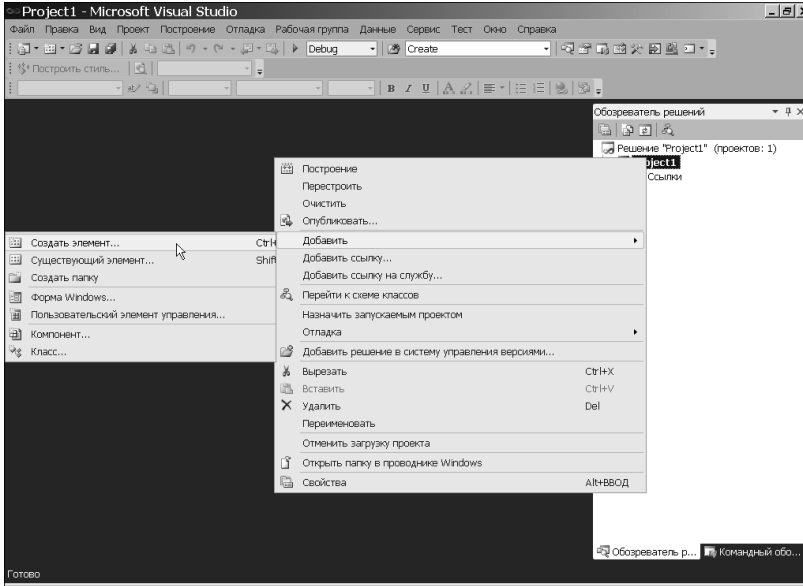
Щелкните на кнопке ОК, чтобы создать проект.

- После создания нового проекта среда Visual Studio будет выглядеть так, как показано на рисунке.

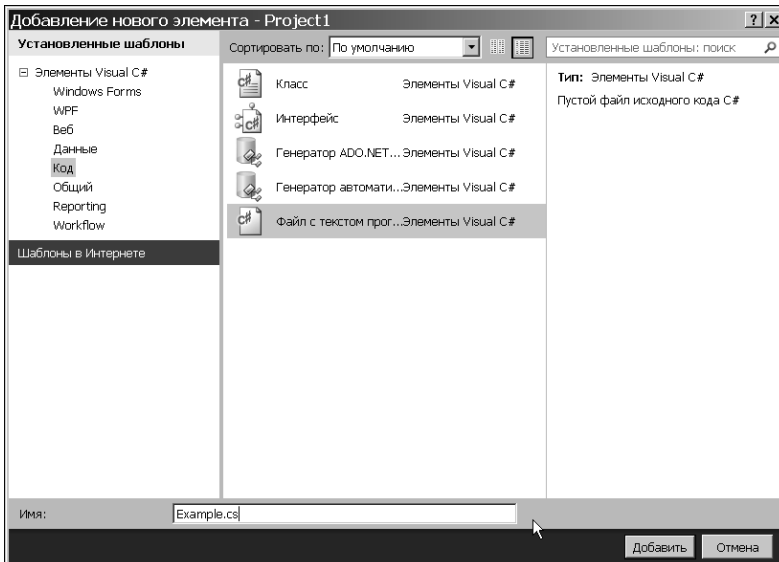


Если по какой-либо причине окно **Обозреватель решений** (Solution Explorer) будет отсутствовать, откройте его с помощью команды **Вид⇒Обозреватель решений** (View⇒Solution Explorer).

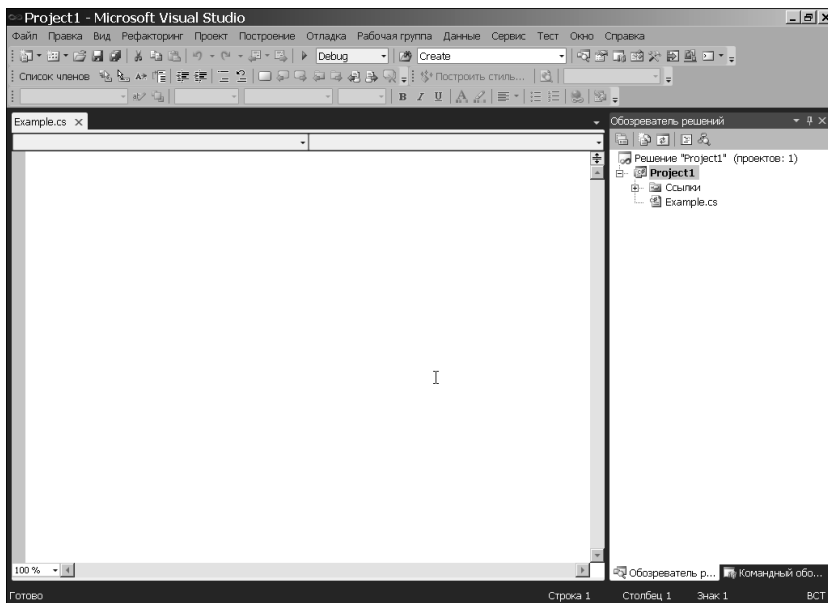
- На данном этапе проект пуст, и вам нужно ввести в него файл с исходным текстом программы на C#. Для этого щелкните правой кнопкой мыши на имени проекта (в данном случае — Project1) в окне **Обозреватель решений**, а затем выберите команду **Добавить (Add)** из контекстного меню. В итоге появится подменю, показанное на рисунке.



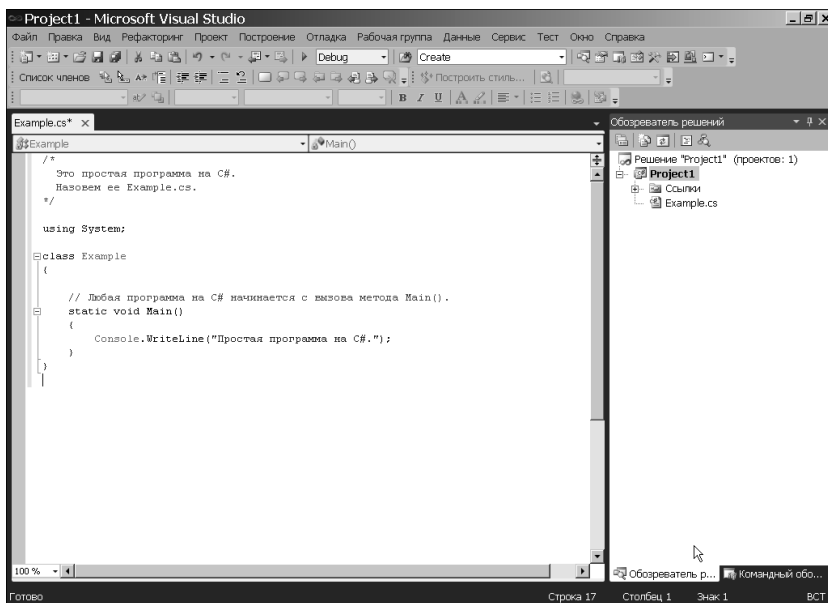
- Выберите команду **Создать элемент (New Item)**, чтобы открыть диалоговое окно **Добавление нового элемента (Add New Item)**. Выберите сначала элемент **Код (Code)** из списка **Установленные шаблоны**, а затем шаблон **Файл с текстом программы (Code File)** и измените имя файла на `Example.cs`, как показано на рисунке.



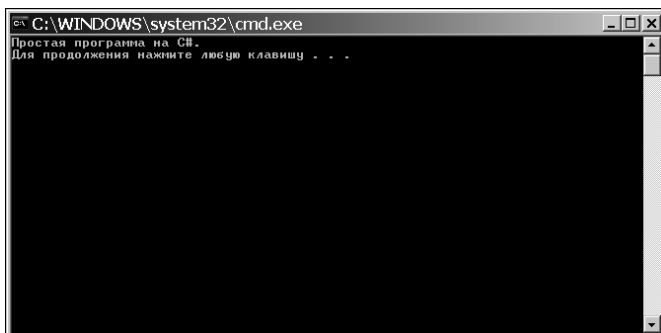
- Введите выбранный файл в проект, щелкнув на кнопке **Добавить**. После этого экран будет выглядеть так, как показано на рисунке.



- Введите исходный текст программы в окне с меткой `Example.cs`, после чего сохраните этот текст в файле. (Исходный текст примеров программ, приведенных в этой книге, можно свободно загрузить по адресу www.mhprofessional.com, чтобы не вводить его каждый раз вручную.) По завершении ввода исходного текста программы экран будет выглядеть так, как показано на рисунке.



7. Скомпилируйте программу, выбрав команду Построение⇒Построить решение (Build⇒Build Solution).
8. Выполните программу, выбрав команду Отладка⇒Запуск без отладки (Debug⇒Start Without Debugging). В результате выполнения программы откроется окно, показанное на рисунке.



Как следует из приведенной выше процедуры, компилирование коротких программ в интегрированной среде разработки требует выполнения немалого числа шагов. Но для каждого примера программы из этой книги вам совсем не обязательно создавать новый проект. Вместо этого вы можете пользоваться одним и тем же проектом C#. С этой целью удалите текущий исходный файл и введите новый. Затем перекомпилируйте и выполните программу. Благодаря этому существенно упрощается весь процесс разработки коротких программ. Однако для разработки реальных приложений каждой программе потребуется отдельный проект.

ПРИМЕЧАНИЕ

Приведенных выше инструкций достаточно для компилирования и выполнения примеров программ, представленных в этой книге, но если вы собираетесь пользоваться Visual Studio как основной средой для разработки программ, вам придется более подробно ознакомиться с ее возможностями и средствами. Это весьма эффективная среда разработки программ, помогающая поддерживать крупные проекты на поддающемся управлению организационном уровне. Данная интегрированная среда разработки позволяет также правильно организовать файлы и связанные с проектом ресурсы. Поэтому целесообразно потратить время и приложить усилия, чтобы приобрести необходимые навыки работы в среде Visual Studio.

Построчный анализ первого примера программы

Несмотря на то что пример программы `Example.cs` довольно краток, в нем демонстрируется ряд ключевых средств, типичных для всех программ на C#. Проанализируем более подробно каждую строку этой программы, начиная с ее имени.

В отличие от ряда других языков программирования, и в особенности Java, где имя файла программы имеет большое значение, имя программы на C# может быть произвольным. Ранее вам было предложено присвоить программе из первого примера имя `Example.cs`, чтобы успешно скомпилировать и выполнить ее, но в C# файл с исходным текстом этой программы можно было бы назвать как угодно. Например, его можно было назвать `Sample.cs`, `Test.cs` или даже `X.cs`.

В файлах с исходным текстом программ на C# условно принято расширение `.cs`, и это условие вы должны соблюдать. Кроме того, многие программисты называют файлы с исходным текстом своих программ по имени основного класса, определенного в программе. Именно поэтому в рассматриваемом здесь примере было выбрано имя файла `Example.cs`. Но поскольку имена программ на C# могут быть произвольными, то они не указываются в большинстве примеров программ, приведенных в настоящей книге. Поэтому вы вольны сами выбирать для них имена.

Итак, анализируемая программа начинается с таких строк.

```
/*
  Это простая программа на C#.
  Назовем ее Example.cs.
*/
```

Эти строки образуют *комментарий*. Как и в большинстве других языков программирования, в C# допускается вводить комментарии в файл с исходным текстом программы. Содержимое комментария игнорируется компилятором. Но, с другой стороны, в комментарии дается краткое описание или пояснение работы программы для всех, кто читает ее исходный текст. В данном случае в комментарии дается описание программы и напоминание о том, что ее исходный файл называется `Example.cs`. Разумеется, в комментариях к реальным приложениям обычно поясняется работа отдельных частей программы или же функции конкретных средств.

В C# поддерживаются три стиля комментариев. Один из них приводится в самом начале программы и называется *многострочным комментарием*. Этот стиль комментария должен начинаться символами `/*` и оканчиваться символами `*/`. Все, что находится между этими символами, игнорируется компилятором. Как следует из его названия, многострочный комментарий может состоять из нескольких строк.

Рассмотрим следующую строку программы.

```
using System;
```

Эта строка означает, что в программе используется пространство имен `System`. В C# *пространство имен* определяет область объявлений. Подробнее о пространстве имен речь пойдет далее в этой книге, а до тех пор поясним вкратце его назначение. Благодаря пространству имен одно множество имен отделяется от других. По существу, имена, объявляемые в одном пространстве имен, не вступают в конфликт с именами, объявляемыми в другом пространстве имен. В анализируемой программе используется пространство имен `System`, которое зарезервировано для элементов, связанных с библиотекой классов среды .NET Framework, применяемой в C#. Ключевое слово `using` просто констатирует тот факт, что в программе используются имена в заданном пространстве имен. (Попутно обратим внимание на весьма любопытную возможность создавать собственные пространства имен, что особенно полезно для работы над крупными проектами.)

Перейдем к следующей строке программы.

```
class Example {
```

В этой строке ключевое слово `class` служит для объявления вновь определяемого класса. Как упоминалось выше, класс является основной единицей инкапсуляции в C#, а `Example` — это имя класса. Определение класса начинается с открывающей фигурной скобки (`{`) и оканчивается закрывающей фигурной скобкой (`}`). Элементы, заключенные в эти фигурные скобки, являются членами класса. Не вдаваясь пока что

в подробности, достаточно сказать, что в C# большая часть действий, выполняемых в программе, происходит именно в классе.

Следующая строка программы содержит *однострочный комментарий*.

```
// Любая программа на C# начинается с вызова метода Main().
```

Это второй стиль комментариев, поддерживаемых в C#. Однострочный комментарий начинается и оканчивается символами //. Несмотря на различие стилей комментариев, программисты нередко пользуются многострочными комментариями для более длинных примечаний и однострочными комментариями для коротких, построчных примечаний к программе. (Третий стиль комментариев, поддерживаемых в C#, применяется при создании документации и описывается в приложении А.)

Перейдем к анализу следующей строки программы.

```
static void Main() {
```

Эта строка начинается с метода `Main()`. Как упоминалось выше, в C# подпрограмма называется методом. И, как поясняется в предшествующем комментарии, именно с этой строки начинается выполнение программы. Выполнение всех приложений C# начинается с вызова метода `Main()`. Разбирать полностью значение каждого элемента данной строки пока что не имеет смысла, потому что для этого нужно знать ряд других средств C#. Но поскольку данная строка используется во многих примерах программ, приведенных в этой книге, то проанализируем ее вкратце.

Данная строка начинается с ключевого слова `static`. Метод, определяемый ключевым словом `static`, может вызываться до создания объекта его класса. Необходимость в этом объясняется тем, что метод `Main()` вызывается при запуске программы. Ключевое слово `void` указывает на то, что метод `Main()` не возвращает значение. В дальнейшем вы узнаете, что методы могут также возвращать значения. Пустые круглые скобки после имени метода `Main` означают, что этому методу не передается никакой информации. Теоретически методу `Main()` можно передать информацию, но в данном примере этого не делается. И последним элементом анализируемой строки является символ `{`, обозначающий начало тела метода `Main()`. Весь код, составляющий тело метода, находится между открывающими и закрывающими фигурными скобками.

Рассмотрим следующую строку программы. Обратите внимание на то, что она находится внутри метода `Main()`.

```
Console.WriteLine("Простая программа на C#.");
```

В этой строке осуществляется вывод на экран текстовой строки "Простая программа на C#.". Сам вывод выполняется встроенным методом `WriteLine()`. В данном примере метод `WriteLine()` выводит на экран строку, которая ему передается. Информация, передаваемая методу, называется *аргументом*. Помимо текстовых строк, метод `WriteLine()` позволяет выводить на экран другие виды информации. Анализируемая строка начинается с `Console` — имени предопределенного класса, поддерживающего ввод-вывод на консоль. Сочетание обозначений `Console` и `WriteLine()` указывает компилятору на то, что метод `WriteLine()` является членом класса `Console`. Применение в C# объекта для определения вывода на консоль служит еще одним свидетельством объектно-ориентированного характера этого языка программирования.

Обратите внимание на то, что оператор, содержащий вызов метода `WriteLine()`, оканчивается точкой с запятой, как, впрочем, и рассматривавшаяся ранее директива `using System`. Как правило, операторы в C# оканчиваются точкой с запятой. Исключением из этого правила служат блоки, которые начинаются символом `{`

и оканчиваются символом `}`. Именно поэтому строки программы с этими символами не оканчиваются точкой с запятой. Блоки обеспечивают механизм группирования операторов и рассматриваются далее в этой главе.

Первый символ `}` в анализируемой программе завершает метод `Main()`, а второй — определение класса `Example`.

И наконец, в C# различаются прописные и строчные буквы. Несоблюдение этого правила может привести к серьезным осложнениям. Так, если вы неумышленно наберете `main` вместо `Main` или же `writeline` вместо `WriteLine`, анализируемая программа окажется ошибочной. Более того, компилятор C# не предоставит возможность выполнить классы, которые не содержат метод `Main()`, хотя и *скомпилирует* их. Поэтому если вы неверно наберете имя метода `Main`, то получите от компилятора сообщение об ошибке, уведомляющее о том, что в исполняемом файле `Example.exe` не определена точка входа.

Обработка синтаксических ошибок

Если вы только начинаете изучать программирование, то вам следует научиться правильно истолковывать (и реагировать на) ошибки, которые могут появиться при попытке скомпилировать программу. Большинство ошибок компиляции возникает в результате опечаток при наборе исходного текста программы. Все программисты рано или поздно обнаруживают, что при наборе исходного текста программы очень легко сделать опечатку. Правда, если вы наберете что-нибудь неправильно, компилятор выдаст соответствующее сообщение о *синтаксической ошибке* при попытке скомпилировать вашу программу. В таком сообщении обычно указывается номер строки исходного текста программы, где была обнаружена ошибка, а также кратко описывается характер ошибки.

Несмотря на всю полезность сообщений о синтаксических ошибках, выдаваемых компилятором, они иногда вводят в заблуждение. Ведь компилятор C# пытается извлечь какой-то смысл из исходного текста, как бы он ни был набран. Именно по этой причине ошибка, о которой сообщает компилятор, не всегда отражает настоящую причину возникшего затруднения. Неумышленный пропуск открывающей фигурной скобки после метода `Main()` в рассмотренном выше примере программы приводит к появлению приведенной ниже последовательности сообщений об ошибках при компиляции данной программы компилятором командной строки `csc`. (Аналогичные ошибки появляются при компиляции в интегрированной среде разработки `Visual Studio`.)

```
EX1.CS (12,21) : ошибка CS1002: ; ожидалось
EX1.CS (13,22) : ошибка CS1519: Недопустимая лексема '(' в
объявлении члена класса, структуры или интерфейса
EX1.CS (15,1) : ошибка CS1022: Требуется определение типа
или пространства имен либо признак конца файла
```

Очевидно, что первое сообщение об ошибке нельзя считать верным, поскольку пропущена не точка с запятой, а фигурная скобка. Два других сообщения об ошибках вносят такую же путаницу.

Из всего изложенного выше следует, что если программа содержит синтаксическую ошибку, то сообщения компилятора не следует понимать буквально, поскольку они могут ввести в заблуждение. Для выявления истинной причины ошибки может

потребуется критический пересмотр сообщения об ошибке. Кроме того, полезно проанализировать несколько строк кода, предшествующих той строке, в которой обнаружена сообщаемая ошибка. Иногда об ошибке сообщается лишь через несколько строк после того места, где она действительно произошла.

Незначительное изменение программы

Несмотря на то что приведенная ниже строка указывается во всех примерах программ, рассматриваемых в этой книге, формально она не нужна.

```
using System;
```

Тем не менее она указывается ради удобства. Эта строка не нужна потому, что в C# можно всегда *полностью определить* имя с помощью пространства имен, к которому оно принадлежит. Например, строку

```
Console.WriteLine("Простая программа на C#.");
```

можно переписать следующим образом.

```
System.Console.WriteLine("Простая программа на C#.");
```

Таким образом, первый пример программы можно видоизменить так.

```
// В эту версию не включена строка "using System;".
```

```
class Example {
    // Любая программа на C# начинается с вызова метода Main().
    static void Main() {
        // Здесь имя Console.WriteLine полностью определено.
        System.Console.WriteLine("Простая программа на C#.");
    }
}
```

Указывать пространство имен `System` всякий раз, когда используется член этого пространства, — довольно утомительное занятие, и поэтому большинство программистов на C# вводят директиву `using System` в начале своих программ, как это сделано в примерах всех программ, приведенных в данной книге. Следует, однако, иметь в виду, что любое имя можно всегда определить, явно указав его пространство имен, если в этом есть необходимость.

Вторая простая программа

В языке программирования, вероятно, нет более важной конструкции, чем переменная. *Переменная* — это именованная область памяти, для которой может быть установлено значение. Она называется переменной потому, что ее значение может быть изменено по ходу выполнения программы. Иными словами, содержимое переменной поддежит изменению и не является постоянным.

В приведенной ниже программе создаются две переменные — `x` и `y`.

```
// Эта программа демонстрирует применение переменных.

using System;

class Example2 {
    static void Main() {
        int x; // здесь объявляется переменная
        int y; // здесь объявляется еще одна переменная

        x = 100; // здесь переменной x присваивается значение 100

        Console.WriteLine("x содержит " + x);

        y = x / 2;

        Console.Write("y содержит x / 2: ");
        Console.WriteLine(y);
    }
}
```

Выполнение этой программы дает следующий результат.

```
x содержит 100
y содержит x / 2: 50
```

В этой программе вводится ряд новых понятий. Прежде всего, в операторе

```
int x; // здесь объявляется переменная
```

объявляется переменная целочисленного типа с именем *x*. В C# все переменные должны объявляться до их применения. Кроме того, нужно обязательно указать тип значения, которое будет храниться в переменной. Это так называемый *тип* переменной. В данном примере в переменной *x* хранится целочисленное значение, т.е. целое число. Для объявления в C# переменной целочисленного типа перед ее именем указывается ключевое слово `int`. Таким образом, в приведенном выше операторе объявляется переменная *x* типа `int`.

В следующей строке объявляется вторая переменная с именем *y*.

```
int y; // здесь объявляется еще одна переменная
```

Как видите, эта переменная объявляется таким же образом, как и предыдущая, за исключением того, что ей присваивается другое имя.

В целом, для объявления переменной служит следующий оператор:

```
тип имя_переменной;
```

где *тип* — это конкретный тип объявляемой переменной, а *имя_переменной* — имя самой переменной. Помимо типа `int`, в C# поддерживается ряд других типов данных.

В следующей строке программы переменной *x* присваивается значение 100.

```
x = 100; // здесь переменной x присваивается значение 100
```

В C# оператор присваивания обозначается одиночным знаком равенства (=). Данный оператор выполняет копирование значения, расположенного справа от знака равенства, в переменную, находящуюся слева от него.

В следующей строке программы осуществляется вывод на экран текстовой строки "x содержит " и значения переменной x.

```
Console.WriteLine("x содержит " + x);
```

В этом операторе знак + обозначает, что значение переменной x выводится вслед за предшествующей ему текстовой строкой. Если обобщить этот частный случай, то с помощью знака операции + можно организовать сцепление какого угодно числа элементов в одном операторе с вызовом метода `WriteLine()`.

В следующей строке программы переменной y присваивается значение переменной x, деленное на 2.

```
y = x / 2;
```

В этой строке значение переменной x делится на 2, а полученный результат сохраняется в переменной y. Таким образом, после выполнения данной строки в переменной y содержится значение 50. При этом значение переменной x не меняется. Как и в большинстве других языков программирования, в C# поддерживаются все арифметические операции, в том числе и перечисленные ниже.

+	Сложение
-	Вычитание
*	Умножение
/	Деление

Рассмотрим две оставшиеся строки программы.

```
Console.Write("y содержит x / 2: ");
Console.WriteLine(y);
```

В этих строках обнаруживаются еще две особенности. Во-первых, для вывода текстовой строки "y содержит x / 2: " на экран используется встроенный метод `Write()`. После этой текстовой строки новая строка не следует. Это означает, что последующий вывод будет осуществлен в той же самой строке. Метод `Write()` подобен методу `WriteLine()`, за исключением того, что после каждого его вызова вывод не начинается с новой строки. И во-вторых, обратите внимание на то, что в вызове метода `WriteLine()` указывается только переменная y. Оба метода, `Write()` и `WriteLine()`, могут быть использованы для вывода значений любых встроенных в C# типов.

Прежде чем двигаться дальше, следует упомянуть еще об одной особенности объявления переменных. Две или более переменных можно указать в одном операторе объявления. Нужно лишь разделить их запятой. Например, переменные x и y могут быть объявлены следующим образом.

```
int x, y; // обе переменные объявляются в одном операторе
```

ПРИМЕЧАНИЕ

В C# внедрено средство, называемое **неявно типизированной переменной**. Неявно типизированными являются такие переменные, тип которых автоматически определяется компилятором. Подробнее неявно типизированные переменные рассматриваются в главе 3.

Другие типы данных

В предыдущем примере программы использовались переменные типа `int`. Но в переменных типа `int` могут храниться только целые числа. Их нельзя использовать в операциях с числами, имеющими дробную часть. Например, переменная типа `int` может содержать значение 18, но не значение 18,3. Правда, `int` — далеко не единственный тип данных, определяемых в C#. Для операций с числами, имеющими дробную часть, в C# предусмотрены два типа данных с плавающей точкой: `float` и `double`. Они обозначают числовые значения с одинарной и двойной точностью соответственно. Из этих двух типов чаще всего используется тип `double`.

Для объявления переменной типа `double` служит оператор

```
double result;
```

где `result` — это имя переменной типа `double`. А поскольку переменная `result` имеет тип данных с плавающей точкой, то в ней могут храниться такие числовые значения, как, например, 122,23, 0,034 или -19,0.

Для лучшего понимания отличий между типами данных `int` и `double` рассмотрим такой пример программы.

```
/*  
  Эта программа демонстрирует отличия  
  между типами данных int и double.  
*/  
  
using System;  
  
class Example3 {  
    static void Main() {  
        int ivar; // объявить целочисленную переменную  
        double dvar; // объявить переменную с плавающей точкой  
  
        ivar = 100; // присвоить переменной ivar значение 100  
  
        dvar = 100.0; // присвоить переменной dvar значение 100.0  
  
        Console.WriteLine("Исходное значение ivar: " + ivar);  
        Console.WriteLine("Исходное значение dvar: " + dvar);  
  
        Console.WriteLine(); // вывести пустую строку  
  
        // Разделить значения обеих переменных на 3.  
        ivar = ivar / 3;  
        dvar = dvar / 3.0;  
  
        Console.WriteLine("Значение ivar после деления: " + ivar);  
        Console.WriteLine("Значение dvar после деления: " + dvar);  
    }  
}
```

Ниже приведен результат выполнения приведенной выше программы.

```
Исходное значение ivar: 100  
Исходное значение dvar: 100
```

Значение `ivar` после деления: 33
 Значение `dvar` после деления: 33.33333333333333

Как видите, при делении значения переменной `ivar` типа `int` на 3 остается лишь целая часть результата — 33, а дробная его часть теряется. В то же время при делении значения переменной `dvar` типа `double` на 3 дробная часть результата сохраняется.

Как демонстрирует данный пример программы, в числовых значениях с плавающей точкой следует использовать обозначение самой десятичной точки. Например, значение 100 в C# считается целым, а значение 100,0 — с плавающей точкой.

В данной программе обнаруживается еще одна особенность. Для вывода пустой строки достаточно вызвать метод `WriteLine()` без аргументов.

Типы данных с плавающей точкой зачастую используются в операциях с реальными числовыми величинами, где обычно требуется дробная часть числа. Так, приведенная ниже программа вычисляет площадь круга, используя значение 3,1416 числа “пи”.

```
// Вычислить площадь круга.

using System;

class Circle {
    static void Main() {
        double radius;
        double area;

        radius = 10.0;
        area = radius * radius * 3.1416;

        Console.WriteLine("Площадь равна " + area);
    }
}
```

Выполнение этой программы дает следующий результат.

Площадь равна 314.16

Очевидно, что вычисление площади круга не дало бы удовлетворительного результата, если бы при этом не использовались данные с плавающей точкой.

Два управляющих оператора

Выполнение программы внутри метода (т.е. в его теле) происходит последовательно от одного оператора к другому, т.е. по цепочке сверху вниз. Этот порядок выполнения программы можно изменить с помощью различных управляющих операторов, поддерживаемых в C#. Более подробно управляющие операторы будут рассмотрены в дальнейшем, а здесь они представлены вкратце, поскольку используются в последующих примерах программ.

Условный оператор

С помощью условного оператора `if` в C# можно организовать выборочное выполнение части программы. Оператор `if` действует в C# практически так же, как и оператор `IF` в любом другом языке программирования. В частности, с точки зрения синтак-

сиса он тождествен операторам `if` в C, C++ и Java. Ниже приведена простейшая форма этого оператора.

```
if(условие) оператор;
```

Здесь *условие* представляет собой булево, т.е. логическое, выражение, принимающее одно из двух значений: “истина” или “ложь”. Если условие истинно, то *оператор* выполняется. А если условие ложно, то выполнение программы происходит, минуя *оператор*. Ниже приведен пример применения условного оператора.

```
if(10 < 11) Console.WriteLine("10 меньше 11");
```

В данном примере условное выражение принимает истинное значение, поскольку 10 меньше 11, и поэтому метод `WriteLine()` выполняется. А теперь рассмотрим другой пример.

```
if(10 < 9) Console.WriteLine("не подлежит выводу");
```

В данном примере 10 не меньше 9. Следовательно, вызов метода `WriteLine()` не произойдет.

В C# определен полный набор операторов отношения, которые можно использовать в условных выражениях. Ниже перечислены все эти операторы и их обозначения.

Операция	Значение
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно

Далее следует пример еще одной программы, демонстрирующей применение условного оператора `if`.

```
// Продемонстрировать применение условного оператора if.

using System;

class IfDemo {
    static void Main() {
        int a, b, c;

        a = 2;
        b = 3;

        if(a < b) Console.WriteLine("a меньше b");

        // Не подлежит выводу.
        if(a == b) Console.WriteLine("этого никто не увидит");

        Console.WriteLine();

        c = a - b; // c содержит -1
```

```

Console.WriteLine("c содержит -1");
if(c >= 0) Console.WriteLine("значение c неотрицательно");
if(c < 0) Console.WriteLine("значение c отрицательно");

Console.WriteLine();

c = b - a; // теперь c содержит 1
Console.WriteLine("c содержит 1");
if(c >= 0) Console.WriteLine("значение c неотрицательно");
if(c < 0) Console.WriteLine("значение c отрицательно ");
}
}

```

Вот к какому результату приводит выполнение данной программы.

a меньше b

c содержит -1
значение c отрицательно

c содержит 1
значение c неотрицательно

Обратите внимание на еще одну особенность этой программы. В строке

```
int a, b, c;
```

три переменные, a, b и c, объявляются списком, разделяемым запятыми. Как упоминалось выше, если требуется объявить две или более переменные одного и того же типа, это можно сделать в одном операторе, разделив их имена запятыми.

Оператор цикла

Для повторного выполнения последовательности операций в программе можно организовать *цикл*. Язык C# отличается большим разнообразием циклических конструкций. Здесь будет рассмотрен оператор цикла `for`. Как и у оператора `if`, у оператора `for` в C# имеются аналоги в C, C++ и Java. Ниже приведена простейшая форма этого оператора.

```
for(инициализация; условие; итерация) оператор;
```

В самой общей форме в части *инициализация* данного оператора задается начальное значение переменной управления циклом. Часть *условие* представляет собой булево выражение, проверяющее значение переменной управления циклом. Если результат проверки истинен, то цикл продолжается. Если же он ложен, то цикл завершается. В части *итерация* определяется порядок изменения переменной управления циклом на каждом шаге цикла, когда он повторяется. Ниже приведен пример краткой программы, демонстрирующей применение оператора цикла `for`.

```
// Продемонстрировать применение оператора цикла for.
```

```
using System;
```

```
class ForDemo {
```

```

static void Main() {
    int count;
    for(count = 0; count < 5; count = count+1)

        Console.WriteLine("Это подсчет: " + count);

    Console.WriteLine("Готово!");
}
}

```

Вот как выглядит результат выполнения данной программы.

```

Это подсчет: 0
Это подсчет: 1
Это подсчет: 2
Это подсчет: 3
Это подсчет: 4
Готово!

```

В данном примере `count` выполняет роль переменной управления циклом. В инициализирующей части оператора цикла `for` задается нулевое значение этой переменной. В начале каждого шага цикла, включая и первый, проверяется условие `count < 5`. Если эта проверка дает истинный результат, то выполняется оператор, содержащий метод `WriteLine()`. Далее выполняется итерационная часть оператора цикла `for`, где значение переменной `count` увеличивается на 1. Этот процесс повторяется до тех пор, пока значение переменной `count` не достигнет величины 5. В этот момент проверка упомянутого выше условия дает ложный результат, что приводит к завершению цикла. Выполнение программы продолжается с оператора, следующего после цикла.

Любопытно, что в программах, профессионально написанных на С#, вы вряд ли увидите итерационную часть оператора цикла в том виде, в каком она представлена в приведенном выше примере программы, т.е. вы редко встретите следующую строку.

```
count = count + 1;
```

Дело в том, что в С# имеется специальный оператор инкремента, выполняющий приращение на 1 значение переменной, или так называемого операнда. Этот оператор обозначается двумя знаками `++`. Используя оператор инкремента, можно переписать приведенную выше строку следующим образом.

```
count++;
```

Таким образом, оператор цикла `for` из приведенного выше примера программы обычно записывается в следующем виде.

```
for(count = 0; count < 5; count++)
```

Опробуйте этот более краткий способ записи итерационной части цикла. Вы сами можете убедиться, что данный цикл выполняется так же, как и прежде.

В С# имеется также оператор декремента, обозначаемый двумя дефисами (`--`). Этот оператор уменьшает значение операнда на 1.

Использование кодовых блоков

Еще одним важным элементом С# является *кодový блок*, который представляет собой группу операторов. Для его организации достаточно расположить операторы

между открывающей и закрывающей фигурными скобками. Как только кодовый блок будет создан, он станет логическим элементом, который можно использовать в любом месте программы, где применяется одиночный оператор. В частности, кодовый блок может служить адресатом операторов `if` и `for`. Рассмотрим следующий оператор `if`.

```
if(w < h) {
    v = w * h;
    w = 0;
}
```

Если в данном примере кода значение переменной `w` меньше значения переменной `h`, то оба оператора выполняются в кодовом блоке. Они образуют внутри кодового блока единый логический элемент, причем один не может выполняться без другого. Таким образом, если требуется логически связать два (или более) оператора, то для этой цели следует создать кодовый блок. С помощью кодовых блоков можно более эффективно и ясно реализовать многие алгоритмы.

Ниже приведен пример программы, в которой кодовый блок служит для того, чтобы исключить деление на ноль.

```
// Прдемонстрировать применение кодового блока.

using System;

class BlockDemo {
    static void Main() {
        int i, j, d;

        i = 5;
        j = 10;

        // Адресатом этого оператора if служит кодовый блок.
        if(i != 0) {
            Console.WriteLine("i не равно нулю");
            d = j / i;
            Console.WriteLine("j / i равно " + d);
        }
    }
}
```

Вот к какому результату приводит выполнение данной программы.

```
i не равно нулю
j / i равно 2
```

В данном примере адресатом оператора `if` служит кодовый блок, а не единственный оператор. Если условие, управляющее оператором `if`, оказывается истинным, то выполняются три оператора в кодовом блоке. Попробуйте задать нулевое значение переменной `i`, чтобы посмотреть, что из этого получится.

Рассмотрим еще один пример, где кодовый блок служит для вычисления суммы и произведения чисел от 1 до 10.

```
// Вычислить сумму и произведение чисел от 1 до 10.

using System;
```

```

class ProdSum {
    static void Main() {
        int prod;
        int sum;
        int i;

        sum = 0;
        prod = 1;

        for(i=1; i <= 10; i++) {
            sum = sum + i;
            prod = prod * i;
        }

        Console.WriteLine("Сумма равна " + sum);
        Console.WriteLine("Произведение равно " + prod);
    }
}

```

Ниже приведен результат выполнения данной программы.

```

Сумма равна 55
Произведение равно 3628800

```

В данном примере внутри кодового блока организуется цикл для вычисления суммы и произведения. В отсутствие такого блока для достижения того же самого результата пришлось бы организовать два отдельных цикла.

И последнее: кодовые блоки не снижают эффективность программ во время их выполнения. Иными словами, наличие символов { и }, обозначающих кодовый блок, никоим образом не замедляет выполнение программы. В действительности применение кодовых блоков, как правило, приводит к повышению быстродействия и эффективности программ, поскольку они упрощают программирование определенных алгоритмов.

Точка с запятой и оформление исходного текста программы

В C# точка с запятой обозначает конец оператора. Это означает, что каждый оператор в отдельности должен оканчиваться точкой с запятой.

Как вы уже знаете, кодовый блок представляет собой набор логически связанных операторов, заключенных в фигурные скобки. Блок не оканчивается точкой с запятой, поскольку он состоит из группы операторов. Вместо этого окончание кодового блока обозначается закрывающей фигурной скобкой.

В C# конец строки не означает конец оператора — о его окончании свидетельствует только точка с запятой. Именно поэтому оператор можно поместить в любой части строки. Например, на языке C# строки кода

```

x = y;
y = y + 1;
Console.WriteLine(x + " " + y);

```

означают то же самое, что и строка кода

```

x = y; y = y + 1; Console.WriteLine(x + " " + y);

```

Более того, составные элементы оператора можно располагать в отдельных строках. Например, следующий фрагмент кода считается в C# вполне допустимым.

```
Console.WriteLine("Это длинная строка вывода" +
    x + y + z +
    "дополнительный вывод");
```

Такое разбиение длинных строк нередко применяется для того, чтобы сделать исходный текст программы более удобным для чтения. Оно помогает также исключить заворачивание слишком длинных строк.

Возможно, вы уже обратили внимание на то, что в предыдущих примерах программ некоторые операторы были набраны с отступом. В C# допускается свободная форма записи. Это означает, что взаимное расположение операторов в строке не имеет особого значения. Но с годами в программировании сложился общепринятый стиль оформления исходного текста программ с отступами, что существенно облегчает чтение этого текста. Именно этому стилю следуют примеры программ в данной книге, что рекомендуется делать и вам. В соответствии с этим стилем следует делать отступ (в виде нескольких пробелов) после каждой открывающей фигурной скобки и возвращаться назад после закрывающей фигурной скобки. А для некоторых операторов даже требуется дополнительный отступ, но об этом речь пойдет далее.

Ключевые слова C#

Основу любого языка программирования составляют его ключевые слова, поскольку они определяют средства, встроенные в этот язык. В C# определены два общих типа ключевых слов: *зарезервированные* и *контекстные*. Зарезервированные ключевые слова нельзя использовать в именах переменных, классов или методов. Их можно использовать только в качестве ключевых слов. Именно поэтому они и называются *зарезервированными*. Их иногда еще называют *зарезервированными словами*, или *зарезервированными идентификаторами*. В настоящее время в версии 4.0 языка C# определено 77 зарезервированных ключевых слов (табл. 2.1).

Таблица 2.1. Ключевые слова, зарезервированные в языке C#

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

Кроме того, в версии C# 4.0 определены 18 контекстных ключевых слов, которые приобретают особое значение в определенном контексте. В таком контексте они выполняют роль ключевых слов, а вне его они могут использоваться в именах других элементов программы, например в именах переменных. Следовательно, контекстные ключевые слова формально не являются зарезервированными. Но, как правило, их следует считать зарезервированными, избегая их применения в любых других целях. Ведь применение контекстного ключевого слова в качестве имени какого-нибудь другого элемента программы может привести к путанице, и поэтому считается многими программистами плохой практикой. Контекстные ключевые слова приведены в табл. 2.2.

Таблица 2.2. Контекстные ключевые слова в C#

add	dynamic	from	get	global
group	into	join	let	orderby
partial	remove	select	set	value
var	where	yield		

Идентификаторы

В C# идентификатор представляет собой имя, присваиваемое методу, переменной или любому другому определяемому пользователем элементу программы. Идентификаторы могут состоять из одного или нескольких символов. Имена переменных могут начинаться с любой буквы алфавита или знака подчеркивания. Далее может следовать буква, цифра или знак подчеркивания. С помощью знака подчеркивания можно повысить удобочитаемость имени переменной, как, например, `line_count`. Но идентификаторы, содержащие два знака подчеркивания подряд, например, `max__value`, зарезервированы для применения в компиляторе. Прописные и строчные буквы в C# различаются. Так, например `myvar` и `MyVar` — это разные имена переменных. Ниже приведены некоторые примеры допустимых идентификаторов.

Test	x	y2	MaxLoad
up	_top	my_var	sample23

Помните, что идентификатор не может начинаться с цифры. Например, `12x` — недействительный идентификатор. Хорошая практика программирования требует выбирать идентификаторы, отражающие назначение или применение именуемых элементов.

Несмотря на то что зарезервированные ключевые слова нельзя использовать в качестве идентификаторов, в C# разрешается применять ключевое слово с предшествующим знаком `@` в качестве допустимого идентификатора. Например, `@for` — действительный идентификатор. В этом случае в качестве идентификатора фактически служит ключевое слово `for`, а знак `@` просто игнорируется. Ниже приведен пример программы, демонстрирующей применение идентификатора со знаком `@`.

```
// Продемонстрировать применение идентификатора со знаком @.

using System;

class IdTest {
    static void Main() {
        int @if; // применение ключевого слова if
```

```

        //в качестве идентификатора
    for(@if = 0; @if < 10; @if++)
        Console.WriteLine("@if равно " + @if);
    }
}

```

Приведенный ниже результат выполнения этой программы подтверждает, что @if правильно интерпретируется в качестве идентификатора.

```

@if равно 0
@if равно 1
@if равно 2
@if равно 3
@if равно 4
@if равно 5
@if равно 6
@if равно 7
@if равно 8
@if равно 9

```

Откровенно говоря, применять ключевые слова со знаком @ в качестве идентификаторов не рекомендуется, кроме особых случаев. Помимо того, знак @ может предшествовать любому идентификатору, но такая практика программирования считается плохой.

Библиотека классов среды .NET Framework

В примерах программ, представленных в этой главе, применялись два встроенных метода: `WriteLine()` и `Write()`. Как упоминалось выше, эти методы являются членами класса `Console`, относящегося к пространству имен `System`, которое определяется в библиотеке классов для среды .NET Framework. Ранее в этой главе пояснялось, что среда C# опирается на библиотеку классов, предназначенную для среды .NET Framework, чтобы поддерживать операции ввода-вывода, обработку строк, работу в сети и графические пользовательские интерфейсы. Поэтому, вообще говоря, C# представляет собой определенное сочетание самого языка C# и стандартных классов .NET. Как будет показано далее, библиотека классов обеспечивает функциональные возможности, являющиеся неотъемлемой частью любой программы на C#. Для того чтобы научиться программировать на C#, нужно знать не только сам язык, но и уметь пользоваться стандартными классами. Различные элементы библиотеки классов для среды .NET Framework рассматриваются в части I этой книги, а в части II — сама библиотека по отдельным ее составляющим.