

# Содержание

---

Об авторе	21
Посвящение	22
Благодарности	22
<b>Введение</b>	<b>23</b>
Об этой книге	23
Глупые предположения	24
Пиктограммы, используемые в книге	25
Источники дополнительной информации	25
Что дальше	26
Ждем ваших отзывов!	27
<b>Часть 1. Основы программирования на C#</b>	<b>29</b>
<b>Глава 1. Ваше первое консольное приложение на C#</b>	<b>31</b>
Компьютерные языки, C# и .NET	31
Что такое программа	32
Что такое C#	32
Что такое .NET	33
Что такое Visual Studio 2017 и Visual C#	34
Создание первого консольного приложения	35
Создание исходной программы	36
Тестовая поездка	40
Заставим программу работать	41
Обзор консольного приложения	43
Каркас программы	43
Комментарии	43
Тело программы	44
Введение в хитрости панели элементов	45
Сохранение кода на панели элементов	45
Повторное использование кода из панели элементов	46
<b>Глава 2. Работа с переменными</b>	<b>47</b>
Объявление переменной	48
Что такое int	48

Правила объявления переменных	49
Вариации на тему <code>int</code>	50
Представление дробных чисел	51
Работа с числами с плавающей точкой	52
Объявление переменной с плавающей точкой	53
Ограничения переменных с плавающей точкой	54
Десятичные числа: комбинация целых чисел и чисел с плавающей точкой	56
Объявление переменных типа <code>decimal</code>	56
Сравнение десятичных и целых чисел, а также чисел с плавающей точкой	56
Логичен ли логический тип	57
Символьные типы	57
Тип <code>char</code>	58
Специальные символы	58
Тип <code>string</code>	59
Что такое тип-значение	60
Сравнение <code>string</code> и <code>char</code>	61
Вычисление високосных лет: <code>DateTime</code>	62
Объявление числовых констант	64
Преобразование типов	65
Позвольте компилятору <code>C#</code> вывести типы данных	66
<b>Глава 3. Работа со строками</b>	69
Неизменяемость строк	70
Основные операции над строками	72
Сравнение строк	72
Проверка равенства: метод <code>Compare()</code>	73
Сравнение без учета регистра	76
Изменение регистра	76
Отличие строк в разных регистрах	77
Преобразование символов строки в символы верхнего или нижнего регистра	77
Цикл по строке	78
Поиск в строках	79
Как искать	79
Пуста ли строка	80

Получение введенной пользователем информации	80
Удаление пробельных символов	80
Анализ числового ввода	81
Обработка последовательности чисел	84
Объединение массива строк в одну строку	86
Управление выводом программы	86
Использование методов Trim() и Pad()	86
Использование метода Concat()	89
Использование метода Split()	91
Форматирование строк	92
StringBuilder: эффективная работа со строками	97
<b>Глава 4. Операторы</b>	<b>99</b>
Арифметика	99
Простейшие операторы	100
Порядок выполнения операторов	100
Оператор присваивания	102
Оператор инкремента	102
Логично ли логическое сравнение	103
Сравнение чисел с плавающей точкой	104
Составные логические операторы	105
Тип выражения	107
Вычисление типа операции	108
Типы при присваивании	110
Перегрузка операторов	110
<b>Глава 5. Управление потоком выполнения</b>	<b>113</b>
Ветвление с использованием if и switch	114
Инструкция if	115
Инструкция else	118
Как избежать else	119
Вложенные инструкции if	120
Конструкция switch	123
Циклы	125
Цикл while	125
Цикл do...while	130
Операторы break и continue	130

Цикл без счетчика	131
Правила области видимости	135
Цикл <code>for</code>	136
Пример	136
Зачем нужны разные циклы	137
Вложенные циклы	138
Оператор <code>goto</code>	139
<b>Глава 6. Глава для коллекционеров</b>	<b>141</b>
<b>Массивы C#</b>	<b>142</b>
Зачем нужны массивы	142
Массив фиксированного размера	143
Массив переменного размера	145
Свойство <code>Length</code>	148
Инициализация массивов	148
Цикл <code>foreach</code>	148
Сортировка массива данных	150
Использование <code>var</code> для массивов	154
Коллекции C#	155
Синтаксис коллекций	156
Понятие <code>&lt;T&gt;</code>	157
Обобщенные коллекции	157
Использование списков	157
Инстанцирование пустого списка	158
Создание списка целых чисел	158
Создание списка для хранения объектов	159
Преобразования списков в массивы и обратно	159
Подсчет количества элементов в списке	159
Поиск в списках	160
Прочие действия со списками	160
Использование словарей	160
Создание словаря	161
Поиск в словаре	161
Итерирование словаря	161
Инициализаторы массивов и коллекций	163
Инициализация массивов	163
Инициализация коллекций	163

Использование множеств	164
Выполнение специфичных для множеств задач	164
Создание множества	165
Добавление элемента в множество	165
Выполнение объединения	166
Пересечение множеств	167
Получение разности	168
Не используйте старые коллекции	169
<b>Глава 7. Работа с коллекциями</b>	<b>171</b>
Обход каталога файлов	171
Использование программы LoopThroughFiles	172
Начало программы	173
Получение начальных входных данных	173
Создание списка файлов	174
Форматирование вывода	175
Вывод в шестнадцатеричном формате	177
Обход коллекций: итераторы	178
Доступ к коллекции: общая задача	179
Использование foreach	181
Обращение к коллекциям как к массивам: индексаторы	182
Формат индексатора	183
Пример программы с использованием индексатора	183
Блок итератора	187
Создание каркаса блока итератора	188
Итерирование дней в месяцах	189
Что же такое коллекция	191
Синтаксис итератора	192
Блоки итераторов произвольного вида и размера	194
<b>Глава 8. Обобщенность</b>	<b>199</b>
Обобщенность в C#	200
Обобщенные классы безопасны	200
Обобщенные классы эффективны	201
Создание собственного обобщенного класса	202
Очередь посылок	203
Очередь с приоритетами	203
Распаковка пакета	208

Метод <code>Main()</code>	210
Написание обобщенного кода	211
И наконец — обобщенная очередь с приоритетами	212
Использование простого необобщенного класса фабрики	215
Незавершенные дела	217
Пересмотр обобщенности	220
Вариантность	221
Контравариантность	221
Ковариантность	223
<b>Глава 9. Эти исключительные исключения</b>	<b>225</b>
Использование механизма исключений для сообщения об ошибках	226
О <code>try</code> -блоках	227
О <code>catch</code> -блоках	228
О <code>finally</code> -блоках	228
Что происходит при генерации исключения	229
Генерация исключений	231
Для чего нужны исключения	232
Исключительный пример	232
Что делает этот пример “исключительным”	234
Трассировка стека	235
Использование нескольких <code>catch</code> -блоков	235
Планирование стратегии обработки ошибок	238
Вопросы, помогающие при планировании	238
Советы по написанию кода с хорошей обработкой ошибок	239
Анализ возможных исключений метода	241
Как выяснить, какие исключения генерируются теми или иными методами	243
Последний шанс перехвата исключения	244
Генерирующие исключения выражения	245
<b>Глава 10. Списки элементов с использованием перечислений</b>	<b>247</b>
Перечисления в реальном мире	248
Работа с перечислениями	249
Использование ключевого слова <code>enum</code>	250
Создание перечислений с инициализаторами	251
Указание типа данных перечисления	252
Создание флагов-перечислений	252
Применение перечислений в конструкции <code>switch</code>	254

<b>Часть 2. Объектно-ориентированное программирование на C#</b>	257
<b>Глава 11. Что такое объектно-ориентированное программирование</b>	259
Объектно-ориентированная концепция № 1: абстракция	260
Процедурные поездки	261
Объектно-ориентированные поездки	261
Объектно-ориентированная концепция № 2: классификация	262
Зачем нужна классификация	263
Объектно-ориентированная концепция № 3: удобные интерфейсы	264
Объектно-ориентированная концепция № 3: управление доступом	265
Поддержка объектно-ориентированных концепций в C#	266
<b>Глава 12. Немного о классах</b>	267
Определение класса и объекта	268
Определение класса	268
Что такое объект	269
Доступ к членам объекта	270
Пример объектно-основанной программы	271
Различие между объектами	273
Работа со ссылками	273
Классы, содержащие классы	275
Статические члены класса	277
Определение константных членов-данных и членов-данных только для чтения	278
<b>Глава 13. Методы</b>	281
Определение и использование метода	282
Использование методов в ваших программах	283
Аргументы метода	291
Передача аргументов методу	291
Передача методу нескольких аргументов	292
Соответствие определений аргументов их использованию	293
Перегрузка методов	294
Реализация аргументов по умолчанию	296
Возврат значений из метода	299
Возврат значения оператором <code>return</code>	300
Определение метода без возвращаемого значения	301
Возврат нескольких значений с использованием кортежей	303

Кортеж с двумя элементами	303
Применение метода <code>Create()</code>	304
Многоэлементные кортежи	304
Создание кортежей более чем с двумя элементами	306
<b>Глава 14. Поговорим об этом</b>	<b>307</b>
Передача объекта в метод	307
Определение методов	309
Определение статического метода	309
Определение метода экземпляра	311
Полное имя метода	313
Обращение к текущему объекту	314
Ключевое слово <code>this</code>	315
Когда <code>this</code> используется явно	316
Что делать при отсутствии <code>this</code>	319
Использование локальных функций	321
<b>Глава 15. Класс: каждый сам за себя</b>	<b>323</b>
Ограничение доступа к членам класса	324
Пример программы с использованием открытых членов	324
Прочие уровни безопасности	327
Зачем нужно управление доступом	328
Методы доступа	329
Пример управления доступом	330
Выводы	334
Определение свойств класса	334
Статические свойства	335
Побочные действия свойств	336
Дайте компилятору написать свойства для вас	337
Методы и уровни доступа	337
Конструирование объектов с помощью конструкторов	338
Конструкторы, предоставляемые <code>C#</code>	338
Замена конструктора по умолчанию	340
Конструирование объектов	341
Непосредственная инициализация объекта	343
Конструирование с инициализаторами	344
Инициализация объекта без конструктора	345



Применение членов с кодом	346
Создание методов с кодом	346
Определение свойств с кодом	347
Определение конструкторов и деструкторов с кодом	347
Определение методов доступа к свойствам с кодом	347
Определение методов доступа к событиям с кодом	348
<b>Глава 16. Наследование</b>	<b>349</b>
Наследование класса	350
Зачем нужно наследование	352
Более сложный пример наследования	353
ЯВЛЯЕТСЯ или СОДЕРЖИТ	356
Отношение ЯВЛЯЕТСЯ	356
Доступ к <code>BankAccount</code> через содержание	357
Отношение СОДЕРЖИТ	358
Когда использовать отношение ЯВЛЯЕТСЯ и когда — СОДЕРЖИТ	359
Поддержка наследования в C#	360
Заменяемость классов	360
Неверное преобразование времени выполнения	361
Избегание неверных преобразований с помощью оператора <code>is</code>	362
Избегание неверных преобразований с помощью оператора <code>as</code>	363
Класс <code>object</code>	363
Наследование и конструктор	365
Вызов конструктора по умолчанию базового класса	365
Передача аргументов конструктору базового класса	366
Указание конкретного конструктора базового класса	368
Обновленный класс <code>BankAccount</code>	369
<b>Глава 17. Полиморфизм</b>	<b>375</b>
Перегрузка унаследованного метода	376
Простейший случай перегрузки метода	376
Различные классы, различные методы	377
Скрытие метода базового класса	377
Вызов методов базового класса	382
Полиморфизм	384
Что неверно в стратегии использования объявленного типа	385
Использование <code>is</code> для полиморфного доступа к скрытому методу	387

Объявление метода виртуальным и перекрытие	388
Получение максимальной выгоды от полиморфизма	391
Визитная карточка класса: метод ToString()	391
Абстракционизм в C#	392
Разложение классов	392
Абстрактный класс: ничего, кроме идеи	397
Как использовать абстрактные классы	398
Создание абстрактных объектов невозможно	400
Опечатывание класса	400
<b>Глава 18. Интерфейсы</b>	<b>403</b>
Что значит МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК	403
Что такое интерфейс	405
Реализация интерфейса	406
Именованые интерфейсы	407
Зачем C# включает интерфейсы	407
Наследование и реализация интерфейса	407
Преимущества интерфейсов	408
Использование интерфейсов	409
Тип, возвращаемый методом	409
Базовый тип массива или коллекции	410
Более общий тип ссылки	410
Использование предопределенных типов интерфейсов C#	411
Пример программы, использующей отношение МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК	411
Создание собственного интерфейса	411
Реализация интерфейса IComparable<T>	413
Сборка воедино	414
Вернемся к Main()	418
Унификация иерархий классов	419
Что скрыто за интерфейсом	421
Наследование интерфейсов	424
Использование интерфейсов для внесения изменений в объектно-ориентированные программы	425
Гибкие зависимости через интерфейсы	426
Абстрактный или конкретный? Когда следует использовать абстрактный класс, а когда — интерфейс	426
Реализация отношения СОДЕРЖИТ с помощью интерфейсов	427

<b>Глава 19. Делегирование событий</b>	429
Звонок домой: проблема обратного вызова	430
Определение делегата	431
Пример передачи кода	433
Делегирование задания	433
Очень простой первый пример	433
Более реальный пример	435
Обзор большего примера	435
Создание приложения	436
Знакомимся с кодом	439
Жизненный цикл делегата	441
Анонимные методы	443
События C#	444
Проектный шаблон Observer	445
Что такое событие. Публикация и подписка	445
Как издатель оповещает о своих событиях	446
Как подписаться на событие	447
Как опубликовать событие	447
Как передать обработчику события дополнительную информацию	449
Рекомендованный способ генерации событий	449
Как наблюдатели “обрабатывают” событие	450
<b>Глава 20. Пространства имен и библиотеки</b>	453
Разделение одной программы на несколько исходных файлов	454
Разделение единой программы на сборки	455
Выполнимый файл или библиотека	455
Сборки	456
Выполнимые файлы	457
Библиотеки классов	458
Объединение классов в библиотеки	458
Создание проекта библиотеки классов	458
Создание автономной библиотеки классов	459
Добавление второго проекта к существующему решению	460
Создание классов для библиотеки	462
Использование тестового приложения	463
Дополнительные ключевые слова для управления доступом	464
internal: строим глазки ЦРУ	465

protected: поделимся с подклассами	467
protected internal: более изошренная защита	469
Размещение классов в пространствах имен	470
Объявление пространств имен	472
Пространства имен и доступ	473
Использование полностью квалифицированных имен	475
<b>Глава 21. Именованные и необязательные параметры</b>	<b>477</b>
Изучение необязательных параметров	478
Ссылочные типы	480
Выходные параметры	481
Именованные параметры	482
Разрешение перегрузки	483
Альтернативные методы возврата значений	483
Работа с переменными out	484
Возврат значений по ссылке	485
<b>Глава 22. Структуры</b>	<b>487</b>
Сравнение структур и классов	488
Ограничения структур	488
Различия типов-значений	489
Когда следует использовать структуры	489
Создание структур	490
Определение базовой структуры	490
Добавление распространенных элементов структур	491
Использование структур как записей	497
Управление отдельной записью	498
Добавление структур в массивы	498
Перекрытие методов	499
<b>Часть 3. Вопросы проектирования на C#</b>	<b>501</b>
<b>Глава 23. Написание безопасного кода</b>	<b>503</b>
Проектирование безопасного программного обеспечения	504
Определение того, что следует защищать	505
Документирование компонентов программы	505
Разложение компонентов на функции	505
Обнаружение потенциальных угроз в функциях	506
Оценка рисков	507

Построение безопасных приложений Windows	507
Аутентификация с использованием входа в Windows	508
Шифрование информации	511
Безопасность развертывания	511
Построение безопасных приложений Web Forms	512
Атаки SQL Injection	513
Уязвимости сценариев	514
Наилучшие методы защиты приложений Web Forms	515
Использование System.Security	517
<b>Глава 24. Обращение к данным</b>	<b>519</b>
Знакомство с System.Data	520
Классы данных и каркас	522
Получение данных	523
Использование пространства имен System.Data	524
Настройка образца схемы базы данных	524
Подключение к источнику данных	525
Работа с визуальными инструментами	531
Написание кода для работы с данными	532
Использование Entity Framework	536
<b>Глава 25. Рыбалка в потоке</b>	<b>541</b>
Где водится рыба: файловые потоки	541
Потоки	542
Читатели и писатели	542
Использование StreamWriter	544
Пример использования потока	545
Как это работает	547
Наконец-то мы пишем!	551
Использование конструкции using	552
Использование StreamReader	556
Еще о читателях и писателях	560
Другие виды потоков	562
<b>Глава 26. Доступ к Интернету</b>	<b>563</b>
Знакомство с System.Net	564
Как сетевые классы вписываются в каркас	565
Использование пространства имен System.Net	567

Проверка состояния сети	567
Загрузка файла из Интернета	569
Отчет по электронной почте	572
Регистрация сетевой активности	574
<b>Глава 27. Создание изображений</b>	<b>579</b>
Знакомство с System.Drawing	580
Графика	580
Перья	581
Кисти	581
Текст	582
Классы рисования и каркас .NET	583
Использование пространства имен System.Drawing	584
Приступая к работе	584
Настройка проекта	585
Обработка счета	586
Создание подключения к событию	587
Рисование доски	588
Запуск новой игры	589
<b>Предметный указатель</b>	<b>591</b>



## Глава 15

# Класс: каждый сам за себя

### В ЭТОЙ ГЛАВЕ...

- » Защита класса
- » Самостоятельная инициализация объекта
- » Определение нескольких конструкторов
- » Конструирование статических членов и членов класса
- » Работа с членами с кодом

**К**ласс должен сам отвечать за свои действия. Так же как микроволновая печь не должна вспыхнуть, объята пламенем, из-за неверного нажатия кнопки, так и класс не должен скончаться (или прикончить программу) при предоставлении некорректных данных.

Чтобы нести ответственность за свои действия, класс должен убедиться в корректности своего начального состояния и в дальнейшем управлять им так, чтобы оно всегда оставалось корректным. *C#* предоставляет для этого все необходимое.

# Ограничение доступа к членам класса

Простые классы определяют все свои члены как `public`. Рассмотрим программу `BankAccount`, которая поддерживает член-данные `balance` для хранения информации о балансе каждого счета. Сделав этот член `public`, вы допускаете любого в святая святых банка, позволяя каждому самому указывать сумму на счету.

Я не знаю ничего о вашем банке, но мой банк и близко не настолько открыт и всегда строго следит за моим счетом, самостоятельно регистрируя каждое снятие денег со счета и вклад на счет. В конце концов, это позволяет уберечься от всяких недоразумений, если вас вдруг подведет память.



ВНИМАНИЕ!

Вы можете решить, что достаточно лишь определить правило, согласно которому никакие другие классы не должны обращаться к члену `balance` непосредственно. Увы, теоретически это, может быть, и так, но на практике такой подход никогда не работает. Да, программисты начинают работу, преисполненные благими намерениями, которые вскоре непонятно куда исчезают под давлением сроков сдачи проекта...

## Пример программы с использованием открытых членов

В приведенной демонстрационной программе класс `BankAccount` объявляет все методы как `public`, в то же время члены-данные `_accountNumber` и `_balance` сделаны `private`. Эта демонстрационная программа некорректна и не будет компилироваться, так как создана исключительно в дидактических целях.

```
// BankAccount - создание банковского счета с использованием
// переменной типа double для хранения баланса счета (она
// объявлена как private, чтобы скрыть баланс от внешнего
// мира)
// Примечание: пока в программу не будут внесены
// исправления, она не будет компилироваться, так как
// метод Main() обращается к private-члену класса
// BankAccount.
using System;

namespace BankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("В текущем состоянии эта " +
                "программа не компилируется.");
        }
    }
}
```



```

// Открытие банковского счета
Console.WriteLine("Создание объекта " +
    "банковского счета");
BankAccount ba = new BankAccount();
ba.InitBankAccount();
// Обращение к балансу при помощи метода Deposit()
// вполне корректно; Deposit() имеет право доступа ко
// всем членам-данным
ba.Deposit(10);
// Непосредственное обращение к члену-данным вызывает
// ошибку компиляции
Console.WriteLine("Здесь вы получите " +
    "ошибку компиляции");
ba._balance += 10;
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}
}

// BankAccount - определение класса, представляющего
// простейший банковский счет
public class BankAccount
{
    private static int _nextAccountNumber = 1000;
    private int _accountNumber;

    // Хранение баланса в виде одной переменной типа double
    private double _balance;

    // Init - инициализация банковского счета с нулевым
    // балансом и с использованием очередного глобального
    // номера
    public void InitBankAccount()
    {
        _accountNumber = ++_nextAccountNumber;
        _balance = 0.0;
    }

    // GetBalance - получение текущего баланса
    public double GetBalance()
    {
        return _balance;
    }

    // Номер счета
    public int GetAccountNumber()
    {
        return _accountNumber;
    }
    public void SetAccountNumber(int accountNumber)
    {
        this._accountNumber = accountNumber;
    }
}

```

```

// Deposit - позволен любой положительный вклад
public void Deposit(double amount)
{
    if (amount > 0.0)
    {
        _balance += amount;
    }
}

// Withdraw - вы можете снять со счета любую сумму, не
// превышающую баланс; метод возвращает реально снятую
// сумму
public double Withdraw(double withdrawal)
{
    if (_balance <= withdrawal)
    {
        withdrawal = _balance;
    }

    _balance -= withdrawal;
    return withdrawal;
}

// GetString - возвращает информацию о состоянии счета в
// виде строки
public string GetString()
{
    string s = String.Format("#{0} = {1:C}",
                             GetAccountNumber(),
                             GetBalance());
    return s;
}
}
}

```

Класс `BankAccount` предоставляет метод `InitBankAccount()` для инициализации членов класса, метод `Deposit()` — для обработки вкладов на счет и метод `Withdraw()` — для снятия денег со счета. Методы `Deposit()` и `Withdraw()` даже обеспечивают выполнение некоторых рудиментарных правил — “нельзя вкладывать отрицательные суммы” и “нельзя снимать больше, чем есть на счету”. Однако в открытой системе, где член-данные `_balance` доступен для внешних методов (под *внешними* подразумеваются методы “в пределах той же программы, но внешние по отношению к классу”), эти правила могут быть нарушены кем угодно. Особенно существенная проблема может возникнуть при разработке больших проектов группами программистов. Это может стать проблемой и для одного человека, поскольку ему свойственно ошибаться.



ЗАПОМНИ!

Хорошо спроектированный код с правилами, выполнение которых проверяет компилятор, значительно снижает количество источников возможных ошибок. Перед тем как идти дальше, обратите внимание

на то, что приведенная демонстрационная программа не будет компилироваться — при такой попытке вы получите сообщение о том, что обращение к члену `DoubleBankAccount.BankAccount._balance` невозможно:

```
'BankAccount.BankAccount._balance' is inaccessible  
due to its protection level.
```

Трудно сказать, зачем компилятор заставили выводить такие скучные сообщения вместо короткого “не лезь к `private`”, но суть именно в этом. Выражение `ba._balance += 10;` оказывается некорректным именно по этой причине — в силу объявления `_balance` как `private` этот член недоступен методу `Main()`, расположенному вне класса `BankAccount`. Замена данного выражения выражением `ba.Deposit(10)` решает возникшую проблему — метод `BankAccount.Deposit()` объявлен как `public`, а потому доступен для метода `Main()`.



ЗАПОМНИ!

Тип доступа по умолчанию — `private`, так что если вы забыли или сознательно пропустили модификатор для некоторого члена, это аналогично тому, как если бы вы описали его как `private`. Однако настоятельно рекомендуется всегда использовать это ключевое слово явно во избежание любых недоразумений. Хороший программист всегда явно указывает свои намерения, что является еще одним методом снижения количества возможных ошибок.

## Прочие уровни безопасности



ВНИМАНИЕ!

В этом разделе используются определенные знания о наследовании и пространствах имен, которые будут рассмотрены в более поздних главах книги (глава 16, “Наследование”, и 20, “Пространства имен и библиотеки”). Вы можете пропустить этот раздел и вернуться к нему позже, получив необходимые знания. Язык `C#` предоставляет следующие уровни безопасности.

- » Члены, объявленные как `public`, доступны любому классу программы.
- » Члены, объявленные как `private`, доступны только из текущего класса.
- » Члены, объявленные как `protected`, доступны только из текущего класса и всех его подклассов.
- » Члены, объявленные как `internal`, доступны для любого класса в том же модуле программы.

*Модулем (module), или сборкой (assembly), в C# называется отдельно компилируемая часть кода, представляющая собой выполняемую*

.EXE-программу либо библиотеку .DLL. Одно пространство имен может распространяться на несколько модулей. (В главе 20, “Пространства имен и библиотеки”, рассматриваются сборки и пространства имен C# и обсуждаются уровни доступа, отличные от `public` и `private`.)

- » Члены, объявленные как `internal protected`, доступны для текущего класса и всех его подклассов, а также классов в том же модуле программы.

Соккрытие членов путем объявления их как `private` обеспечивает максимальную степень безопасности. Однако зачастую такая высокая степень и не нужна. В конце концов, члены подклассов и так зависят от членов базового класса, так что ключевое слово `protected` предоставляет достаточно удобный уровень безопасности.

## Зачем нужно управление доступом

Объявление внутренних членов класса как `public` — не лучшая мысль как минимум по следующим причинам.

- » **Объявляя члены-данные `public`, вы не в состоянии просто определить, когда и как они модифицируются.** Зачем беспокоиться и создавать методы `Deposit()` и `Withdraw()` с проверками корректности? И вообще, зачем создавать любые методы, ведь любой метод любого класса может модифицировать данные счета в любой момент. Но если другой метод может обращаться к этим данным, то он практически обязательно это сделает.  
Ваша программа `BankAccount` может проработать длительное время, прежде чем вы заметите, что баланс одного из счетов отрицателен. Метод `Withdraw()` призван оградить от подобной ситуации, но в описанном случае непосредственный доступ к балансу, минуя метод `Withdraw()`, имеют и другие методы. Вычислить, какие именно методы и при каких условиях поступают так некорректно, — задача не из легких.
- » **Доступ ко всем членам-данным класса делает его интерфейс слишком сложным.** Как программист, использующий класс `BankAccount`, вы не хотите знать о том, что делается внутри него. Вам достаточно знаний о том, как положить деньги на счет и снять их с него.
- » **Доступ ко всем членам-данным класса приводит к “растеканию” правил класса.** Например, класс `BankAccount` не позволяет баланс стать отрицательным ни при каких условиях. Это —

*бизнес-правило*, которое должно быть локализовано в методе `Withdraw()`. В противном случае вам придется добавлять соответствующую проверку в весь код, в котором осуществляется изменение баланса.

Что произойдет, если банк решит изменить правила и часть клиентов с хорошей кредитной историей получит право на небольшой отрицательный баланс в течение короткого времени? Вам придется долго рыскать по всей программе и вносить изменения во все места, где выполняется непосредственное обращение к балансу.



СОВЕТ

Не делайте классы и методы более доступными, чем это необходимо. Это не параноидальная боязнь хакеров — это просто поможет вам снизить количество ошибок в коде. По возможности используйте модификатор `private`, а затем при необходимости поднимайте его до `protected`, `internal`, `internal protected` или `public`.

## Методы доступа

Если вы более внимательно посмотрите на класс `BankAccount`, то увидите несколько других методов. Один из них, `GetString()`, возвращает строковую версию счета для вывода ее на экран посредством вызова `Console.WriteLine()`. Дело в том, что вывод содержимого объекта `BankAccount` может быть затруднен, если это содержимое недоступно. К тому же, следуя принципу “отдайте кесарю кесарево”, класс должен иметь право сам решать, как он будет представлен при выводе.

Кроме того, имеется два метода для *получения значения*, `GetBalance()` и `GetAccountNumber()`, и метод *установки значения* — `SetAccountNumber()`. Вы можете удивиться: зачем так волноваться из-за того, что член `_balance` будет объявлен как `private`, и при этом предоставлять метод `GetBalance()`? На самом деле для этого имеются достаточно веские основания.

- » **GetBalance()** не дает возможности изменять член `_balance` — он только возвращает его значение. Тем самым значение баланса делается доступным только для чтения. Используя аналогию с настоящим банком, вы можете просмотреть состояние своего счета в любой момент, но не можете снять с него деньги иначе, чем с применением процедур, предусмотренных для этого банком.
- » **Метод GetBalance()** скрывает внутренний формат класса от внешних методов. Метод `GetBalance()` может в процессе работы выполнять некоторые вычисления, обращаться к базе данных банка — словом, выполнять какие-то действия, чтобы получить состояние счета. Внешние методы ничего об этом не знают и не должны знать. Продолжая аналогию, вы интересуетесь состоянием счета, но не знаете, как, где и в каком именно виде хранятся ваши деньги.

И наконец, метод `GetBalance()` предоставляет механизм для внесения внутренних изменений в класс `BankAccount`, абсолютно не затрагивая при этом его пользователей. Если от национального банка придет распоряжение хранить деньги как-то иначе, это никак не должно сказаться на вашем способе обращения со счетом.

## Пример управления доступом

Приведенная далее демонстрационная программа `DoubleBankAccount` указывает потенциальные изъяны программы `BankAccount`. В листинге показан только метод `Main()` — единственная претерпевшая изменения часть программы:

```
// DoubleBankAccount - создание банковского счета с
// использованием переменной типа double для хранения
// баланса счета (она объявлена как private, чтобы скрыть
// баланс от внешнего мира)
using System;
namespace DoubleBankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Открытие банковского счета
            Console.WriteLine("Создание объекта " +
                "банковского счета");
            BankAccount ba = new BankAccount();
            ba.InitBankAccount();

            // Вклад на счет
            double deposit = 123.454;
            Console.WriteLine("Depositing {0:C}", deposit);
            ba.Deposit(deposit);

            // Баланс счета
            Console.WriteLine("Счет = {0}", ba.GetString());

            // Вот где возникает проблема
            double fractionalAddition = 0.002;
            Console.WriteLine("Adding {0:C}", fractionalAddition);
            ba.Deposit(fractionalAddition);

            // Результат
            Console.WriteLine("В результате счет = {0}",
                ba.GetString());

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                "завершения программы...");
            Console.Read();
        }
    }
}
```

Метод `Main()` создает банковский счет и вносит на него сумму 123,454, т.е. сумму с дробным количеством копеек. Затем метод `Main()` вносит на счет еще одну долю копейки и выводит баланс счета. Вывод программы выглядит следующим образом:

```
Создание объекта банковского счета
Вклад $123.45
Счет = #1001 = $123.45
Вклад $0.00
В результате счет = #1001 = $123.46
Нажмите <Enter> для завершения программы...
```

Пользователь начинает жаловаться на некорректные расчеты. Похоже, в программе имеется ошибка.

Проблема, конечно, в том, что 123.454 выводится как 123.45. Чтобы избежать проблем, банк принимает решение округлять вклады и снятия до ближайшей копейки. Простейший путь осуществить это — конвертировать счета в `decimal` и использовать метод `Decimal.Round()`, как это сделано в демонстрационной программе `DecimalBankAccount`.

```
// DecimalBankAccount - создание банковского счета с
// использованием переменной типа decimal для хранения
// баланса счета
using System;

namespace DecimalBankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Открытие банковского счета
            Console.WriteLine("Создание объекта " +
                "банковского счета");
            BankAccount ba = new BankAccount();
            ba.InitBankAccount();

            // Вклад на счет
            double deposit = 123.454;
            Console.WriteLine("Вклад {0:C}", deposit);
            ba.Deposit(deposit);

            // Баланс счета
            Console.WriteLine("Счет = {0}", ba.GetString());

            // Добавляем очень малую величину
            double fractionalAddition = 0.002;
            Console.WriteLine("Вклад {0:C}", fractionalAddition);
            ba.Deposit(fractionalAddition);
        }
    }
}
```

```

        // Результат
        Console.WriteLine("В результате счет = {0}",
            ba.GetString());

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");
        Console.Read();
    }
}

// BankAccount - определение класса, представляющего
// простейший банковский счет
public class BankAccount
{
    private static int _nextAccountNumber = 1000;
    private int _accountNumber;

    // Хранение баланса в виде одной переменной типа decimal
    private decimal _balance;

    // Init - инициализация банковского счета с нулевым
    // балансом и использованием очередного глобального
    // номера
    public void InitBankAccount()
    {
        _accountNumber = ++_nextAccountNumber;
        _balance = 0;
    }

    // GetBalance - получение текущего баланса
    public double GetBalance()
    {
        return (double)_balance;
    }

    // AccountNumber
    public int GetAccountNumber()
    {
        return _accountNumber;
    }
    public void SetAccountNumber(int accountNumber)
    {
        this._accountNumber = accountNumber;
    }

    // Deposit - позволен любой положительный вклад
    public void Deposit(double amount)
    {
        if (amount > 0.0)
        {
            // Округление до ближайшей копейки перед
            // внесением вклада

```



```

        decimal temp = (decimal)amount;
        temp = Decimal.Round(temp, 2);
        _balance += temp;
    }
}

// Withdraw - вы можете снять со счета любую сумму, не
// превышающую баланс; метод возвращает реально снятую
// сумму
public double Withdraw(double withdrawal)
{
    // Преобразуем в тип decimal и работаем с ним.
    decimal decWithdrawal = (decimal)withdrawal;

    if (_balance <= decWithdrawal)
    {
        decWithdrawal = _balance;
    }

    _balance -= decWithdrawal;
    return (double)decWithdrawal; // Возврат double
}

// GetString - возвращает информацию
// о состоянии счета в виде строки
public string GetString()
{
    string s = String.Format("#{0} = {1:C}",
                             GetAccountNumber(),
                             GetBalance());

    return s;
}
}
}

```

Внутреннее представление поменялось на использование значений типа `decimal`, который в любом случае более подходит для работы с банковским счетом, чем тип `double`. Метод `Deposit()` теперь применяет метод `Decimal.Round()` для округления вкладываемой суммы до ближайшей копейки. Вывод программы оказывается таким, как и ожидалось:

```

Создание объекта банковского счета
Вклад $123.45
Счет = #1001 = $123.45
Вклад $0.00
В результате счет = #1001 = $123.45
Нажмите <Enter> для завершения программы...

```

## Выводы

Вы можете сказать, что нужно было с самого начала писать программу `BankAccount` с использованием `decimal`, и, пожалуй, с вами можно согласиться. Но дело не в этом. Могут быть разные приложения и ситуации. Главное, что класс `BankAccount` оказался в состоянии решить проблему так, что не пришлось вносить никаких изменений в использующую его программу (обратите внимание на то, что открытый интерфейс класса не изменился: методы `Balance()` и `Withdraw()` так и возвращают значения типа `double`, а `Deposit()` и `Withdraw()` принимают параметр типа `double`).

В данном случае единственным методом, на который потенциально влияло изменение при непосредственном обращении к балансу, является метод `Main()`, но в реальной программе могут существовать десятки таких методов, и они могут оказаться в не меньшем количестве модулей. В данном случае ни один из этих методов не должен изменяться, потому что исправление находится в пределах класса `BankAccount`, *открытый интерфейс* которого (его открытые методы) не изменился. Если бы методы обращались ко внутренним членам класса непосредственно, это было бы решительно невозможно.



ВНИМАНИЕ!

Внесение внутренних изменений в класс требует определенного тестирования использующего класс кода, несмотря на то, что в него не вносятся никакие модификации.

## Определение свойств класса

Методы `GetX()` и `SetX()`, продемонстрированные в программе `BankAccount`, называются *методами доступа* (access methods). Хотя их использование теоретически является хорошей привычкой, на практике это зачастую приводит к грустным результатам. Судите сами — чтобы увеличить член `_accountNumber` на 1, требуется писать следующий код:

```
SetAccountNumber(GetAccountNumber() + 1);
```

`C#` имеет конструкцию, называемую *свойством* и делающую использование методов доступа существенно более простым. Приведенный далее фрагмент кода определяет свойство `AccountNumber` доступным для чтения и записи:

```
public int AccountNumber          // Скобки не нужны
{
    get{return accountNumber;} // Фигурные скобки и точка с запятой
    set{accountNumber = value;} // Здесь 'value' - ключевое слово
}
```

Раздел `get` реализуется при чтении свойства, а `set` — при записи. В приведенном далее фрагменте исходного текста свойство `Balance` является свойством только для чтения, так как здесь определен только раздел `get`:

```
public double Balance
{
    get
    {
        return (double)balance;
    }
}
```

Использование свойств выглядит следующим образом:

```
BankAccount ba = new BankAccount();
// Записываем свойство AccountNumber

ba.AccountNumber = 1001;
// Считываем оба свойства
Console.WriteLine("#{0} = {1:C}",ba.AccountNumber, ba.Balance);
```

Свойства `AccountNumber` и `Balance` очень похожи на открытые члены-данные как внешне, так и в использовании. Однако свойства позволяют классу защитить свои внутренние члены (так, член `_balance` остается при этом `private`). Обратите внимание, что `Balance` выполняет приведение типа — точно так же может производиться любое количество вычислений. Свойства вовсе не обязательно должны представлять собой одну строку кода и могут выполнять различные действия наподобие проверки входных данных.



СОВЕТ

По соглашению имена свойств начинаются с прописной буквы. Обратите также внимание, что свойства не имеют скобок: следует писать просто `Balance`, а не `Balance()`.



ТЕХНИЧЕСКИЕ  
ПОДРОБНОСТИ

Свойства совсем не обязательно неэффективны. Компилятор `C#` может оптимизировать простой метод доступа так, что он будет генерировать не больше машинных команд, чем непосредственное обращение к члену. Это важно не только для прикладных программ, но и для самого `C#`. Библиотека `C#` широко использует свойства, и то же самое должны делать и вы, даже для обращения к членам-данным класса из методов этого же класса.

## Статические свойства

Статические члены-данные (класса) могут быть доступны через статические свойства, как показано в следующем простейшем примере:

```
public class BankAccount
{
    private static int nextAccountNumber = 1000;
    public static int NextAccountNumber
    {
        get{return nextAccountNumber;}
    }
    // ...
}
```

Свойство `NextAccountNumber` доступно посредством указания имени его класса, так как оно не является свойством конкретного объекта (оно объявлено как `static`).

```
// считываем свойство NextAccountNumber
int value = BankAccount.NextAccountNumber;
```

(В этом примере `value` находится вне контекста свойства, а потому не рассматривается как зарезервированное слово.)

## Побочные действия свойств

Операция `get` может применяться не только для простого получения значения, связанного со свойством. Взгляните на следующий код:

```
public static int AccountNumber
{
    // Получение значения переменной и увеличение ее значения,
    // чтобы в следующий раз получить уже новое ее значение
    get{return ++_nextAccountNumber;}
}
```

Данное свойство увеличивает статический член класса перед тем, как вернуть результат. Однако это не слишком хорошая идея, ведь пользователь ничего не знает о такой особенности и не подозревает, что происходит что-то помимо чтения значения. Увеличение переменной в данном случае представляет собой *побочное действие*.



ЗАПОМНИ!

Подобно методам доступа, которые они имитируют, свойства не должны изменять состояния класса иначе чем через установку значения соответствующего члена данных. В общем случае и свойства, и методы должны избегать побочных действий, так как это может привести к трудноуловимым ошибкам. Изменяйте класс настолько явно и непосредственно, насколько это возможно.

## Дайте компилятору написать свойства для вас

Большинство свойств, описанных в предыдущем разделе, представляют собой простые подпрограммы, писать которые очень просто... и утомительно:

```
private string _name;    // Член, соответствующий свойству
public string Name { get { return _name; } set { _name = value; } }
```

Поскольку код везде оказывается одним и тем же, было решено позволить компилятору C# 3.0 делать эту работу вместо вас. Вот все, что вы должны написать:

```
public string Name { get; set; }
```

Это эквивалентно

```
private string <somename>;    // Что такое <somename>?
                               // неизвестно и неважно.
public string Name { get { return <somename>; }
                    set { <somename> = value; } }
```

Компилятор создает некий загадочный член-данные, который во всем приведенном коде оказывается безымянным. Такой стиль заставляет использовать свойства даже внутри других членов того же класса просто потому, что все, что вам известно, — это имя свойства. По этой причине вы должны иметь оба свойства — и `get`, и `set`. Инициализировать их можно при помощи следующего синтаксиса:

```
public int AnInt {get; set;} // Компилятор создает
                             // закрытую переменную
. . .
AnInt = 2;                   // Инициализация созданной компилятором
                             // переменной при помощи свойства.
```

## Методы и уровни доступа

Методы доступа не обязательно должны быть объявлены как `public`. Вы можете объявлять их на любом уровне доступа, включая `private`, если метод доступа предназначен для использования исключительно внутри собственного класса.

Можно даже отдельно изменять уровень доступа для частей `get` и `set`. Предположим, например, что вы не хотите давать возможность работы с методом `set` вне класса. В этом случае свойство можно записать таким образом:

```
internal string Name { get; private set; }
```

# Конструирование объектов с помощью конструкторов



ЗАПОМНИ!

Управление доступом — это только половина проблемы. Рождение объекта — один из самых важных этапов в его жизни. Класс, конечно, может предоставить метод для инициализации вновь созданного объекта, но беда в том, что приложение может попросту забыть его вызвать. В таком случае члены-данные класса окажутся заполненными “мусором”, и корректной работы от такого объекта ждать не придется. Язык C# решает эту проблему путем вызова инициализирующего метода автоматически, например:

```
MyObject mo = new MyObject();
```

Эта инструкция не только выделяет память для объекта, но и выполняет инициализацию его членов.



ЗАПОМНИ!

Не путайте термины *класс* и *объект*. *Cat* — это класс, но экземпляр класса *Cat* по имени *Striper* — это объект класса *Cat*.

## Конструкторы, предоставляемые C#

Язык C# хорошо умеет отслеживать инициализацию переменных и не позволяет использовать неинициализированные переменные. Например, представленный далее код приведет к генерации ошибки времени компиляции:

```
public static void Main(string[] args)
{
    int n;
    double d;
    double calculatedValue = n + d;
}
```

Язык C# отслеживает тот факт, что ни *n*, ни *d* не имеют присвоенного значения и не могут использоваться в выражении. Компиляция этой микропрограммы приводит к генерации следующих сообщений об ошибках:

```
Use of unassigned local variable 'n'
Use of unassigned local variable 'd'
```

C# предоставляет конструктор по умолчанию, который инициализирует члены данных объекта:

- » числа — нулями;
- » логические переменные — значениями false;
- » ссылки на объекты — значениями null.

Рассмотрим следующую простую демонстрационную программу:

```
using System;
namespace Test
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Сначала создаем объект
            MyObject localObject = new MyObject();
            Console.WriteLine("localObject.n = {0}",
                localObject.n);
            if (localObject.nextObject == null)
            {
                Console.WriteLine("localObject.nextObject = null");
            }
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                "завершения программы...");
            Console.Read();
        }
    }
    public class MyObject
    {
        internal int n;
        internal MyObject nextObject;
    }
}
```

Эта программа определяет класс `MyObject`, который содержит переменную `n` типа `int` и ссылку на объект `nextObject`, позволяющую создавать связанные списки объектов. Метод `Main()` создает объект класса `MyObject` и выводит начальное содержимое его членов. Вывод этой программы имеет следующий вид:

```
localObject.n = 0
localObject.nextObject = null
Нажмите <Enter> для завершения программы...
```

Язык `C#` при создании объекта выполняет небольшой код по инициализации объекта и его членов. Если бы не этот код, члены-данные `localObject.n` и `localObject.nextObject` содержали бы какие-то случайные значения, попросту говоря — “мусор”.



ЗАПОМНИ!

Код, инициализирующий значения при создании, называется *конструктором по умолчанию*. Он “конструирует” класс в смысле инициализации его членов. Таким образом, C# гарантирует, что объект начинает жизнь в известном состоянии — полностью обнуленным. *Это относится только к данным-членам класса, но не к локальным переменным метода.*

## Замена конструктора по умолчанию

Хотя компилятор автоматически инициализирует все переменные экземпляров соответствующими значениями, для многих классов (возможно, даже для большинства) значения по умолчанию не являются корректным состоянием. Рассмотрим класс `BankAccount`, о котором уже шла речь в этой главе.

```
public class BankAccount
{
    private int _accountNumber;
    private double _balance;
    // ... прочие члены
}
```

Хотя нулевое начальное значение баланса вполне корректно, нулевое значение номера счета, определенно, не является верным.

Поэтому в данный момент класс `BankAccount` включает метод `InitBankAccount()`, инициализирующий объект. Однако такой подход перекладывает слишком большую ответственность на прикладную программу, использующую данный класс. Если вдруг приложение забудет вызвать метод `InitBankAccount()`, то прочие методы банковского счета могут оказаться неработоспособными, хотя при этом и не будут содержать никаких ошибок.



ЗАПОМНИ!

Класс не должен полагаться на внешние методы наподобие метода `InitBankAccount()`, которые должны обеспечивать корректное состояние его объектов. Для решения данной проблемы класс предоставляет специальный метод, автоматически вызываемый C# при создании объекта, — *конструктор класса*. Конструктор мог бы именоваться как `Init()`, `Start()` или `Create()`, но C# требует, чтобы конструктор носил то же имя, что и имя самого класса, так что конструктор класса `BankAccount` имеет следующий вид:

```
public void Main(string[] args)
{
    BankAccount ba = new BankAccount(); // Вызов конструктора
}
```



```

public class BankAccount
{
    // Номера банковских счетов начинаются с 1000 и
    // назначаются последовательно в возрастающем порядке
    static int nextAccountNumber = 1000;
    // Для каждого счета поддерживаются его номер и баланс
    int accountNumber;
    double balance;
    // Конструктор BankAccount - обратите внимание на его имя
    public BankAccount() // Требуются круглые скобки, могут
                        // иметься аргументы, возвращаемый
                        // тип отсутствует
    {
        accountNumber = ++nextAccountNumber;
        balance = 0.0;
    }
    // . . . прочие члены . . .
}

```

Содержимое конструктора `BankAccount` то же, что и первоначального метода `InitBankAccount()`. Однако конструктор имеет некоторые особенности:

- » всегда имеет то же имя, что и сам класс;
- » может как принимать параметры, так и вызываться без них;
- » не имеет возвращаемого типа, даже типа `void`;
- » метод `Main()` не должен вызывать никаких дополнительных методов для инициализации объекта при его создании — не нужны никакие вызовы `Init()`.



ЗАПОМНИ!

Если вы создаете собственный конструктор, `C#` не создает конструктор по умолчанию автоматически. Ваш конструктор заменяет конструктор по умолчанию и становится единственным способом создания экземпляра класса.

## Конструирование объектов

Теперь посмотрим на конструкторы в деле. Для этого рассмотрим программу `DemonstrateCustomConstructor`.

```

using System;
// DemonstrateCustomConstructor -- демонстрация работы
// конструкторов по умолчанию; создаем класс с конструктором
// и рассматриваем несколько сценариев.
namespace DemonstrateCustomConstructor
{
    // MyObject - создание класса с "многословным"
    // конструктором и внутренним объектом
    public class MyObject
    {

```

```

// Этот член-данные является свойством класса
private static MyOtherObject _staticObj =
    new MyOtherObject();

// Этот член-данные является свойством каждого объекта
private MyOtherObject _dynamicObj;

// Конструктор (с обильным выводом на экран)
public MyObject()
{
    Console.WriteLine("Начало конструктора MyObject");
    Console.WriteLine(" (Статические члены-данные " +
        "конструируются до этого " +
        "конструктора)");
    Console.WriteLine("Теперь динамически создаем " +
        "нестатический член-данные:");
    _dynamicObj = new MyOtherObject();
    Console.WriteLine("MyObject constructor ending");
}

// MyOtherObject - у этого класса тоже многословный
// конструктор, но внутренние члены-данные отсутствуют
public class MyOtherObject
{
    public MyOtherObject()
    {
        Console.WriteLine("Конструирование MyOtherObject");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Начало метода Main()");
        Console.WriteLine("Создание локального объекта " +
            "MyObject в Main():");
        MyObject localObject = new MyObject();

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");
        Console.Read();
    }
}
}

```

Выполнение данной программы приводит к следующему выводу на экран:

```

Начало метода Main()
Создание локального объекта MyObject в Main():
Конструирование MyOtherObject
Начало конструктора MyObject

```

(Статические члены-данные конструируются до этого конструктора)  
Теперь динамически создаем нестатический член-данные:  
Конструирование `MyOtherObject`  
Завершение конструктора `MyObject`  
Нажмите <Enter> для завершения программы...

Вот реконструкция происходящего при запуске программы.

- 1. Программа начинает работу, и метод `Main()` выводит начальное сообщение и сообщение о предстоящем создании локального объекта `MyObject`.**
- 2. Метод `Main()` создает объект `localObject` типа `MyObject`.**
- 3. `MyObject` содержит статический член `_staticObj` класса `MyOtherObject`.**  
*Все статические члены-данные инициализируются до первого выполнения конструктора `MyObject()`. В этом случае C# присваивает переменной `_staticObj` ссылку на вновь созданный объект перед тем, как передать управление конструктору `MyObject`.*
- 4. Конструктор `MyObject` получает управление. Он выводит начальное сообщение и напоминает, что статический член уже сконструирован до того, как начал работу конструктор `MyObject()`.**
- 5. После объявления о своих намерениях по динамическому созданию нестатического члена конструктор `MyObject` создает объект класса `MyOtherObject` с использованием оператора `new`, что сопровождается выводом второго сообщения о создании `MyOtherObject` на экран.**
- 6. Управление возвращается конструктору `MyObject`, который, в свою очередь, возвращает управление методу `Main()`.**

## Непосредственная инициализация объекта

Помимо инициализации членов-данных в конструкторе, C# позволяет инициализировать члены-данные непосредственно с использованием инициализаторов. Это означает, что класс `BankAccount` можно записать следующим образом:

```
public class BankAccount
{
    // Номера банковских счетов начинаются с 1000 и
    // назначаются последовательно в возрастающем порядке
    static int _nextAccountNumber = 1000;

    // Для каждого счета поддерживаются его номер и баланс
    int _accountNumber = ++_nextAccountNumber;
    double _balance = 0.0;

    // ... прочие члены ...
}
```

Вот в чем состоит работа инициализаторов. Как `_accountNumber`, так и `_balance` получают значения как часть объявления, эффект которого аналогичен использованию указанного кода в конструкторе.

Надо очень четко представлять себе картину происходящего. Вы можете решить, что это выражение присваивает значение `0.0` переменной `_balance` непосредственно. Но ведь `_balance` существует только как часть некоторого объекта. Таким образом, присваивание не выполняется до тех пор, пока не будет создан объект `_BankAccount`. Рассматриваемое присваивание осуществляется всякий раз при создании объекта.

Заметим, что статический член-данные `_nextAccountNumber` инициализируется при первом обращении к классу `BankAccount` (как вы убедились при выполнении демонстрационной программы в отладчике), т.е. при обращении к любому свойству или методу объекта, владеющему статическим членом, в том числе к конструктору.



ЗАПОМНИ!

Будучи инициализированным, статический член повторно не инициализируется, сколько бы объектов вы ни создавали. Этим он отличается от нестатических членов. Инициализаторы выполняются в порядке их появления в объявлении класса. Если C# встречает и инициализаторы, и конструктор, то инициализаторы выполняются раньше тела конструктора.

## Конструирование с инициализаторами

Давайте в программе `DemonstrateCustomConstructor` перенесем вызов `new MyOtherObject()` из конструктора `MyObject` в объявление так, как показано в приведенном далее фрагменте исходного текста полужирным шрифтом, и изменим второй вызов `WriteLine()`.

```
public class MyObject
{
    // Этот член является свойством класса
    private static MyOtherObject _staticObj = new MyOtherObject();

    // Этот член является свойством объекта
    private MyOtherObject _dynamicObj = new MyOtherObject();

    public MyObject()
    {
        Console.WriteLine("Начало конструктора MyObject");
        Console.WriteLine(" (Статические члены " +
            "инициализированы до конструктора) ");
        // Ранее здесь создавался _dynamicObj
        Console.WriteLine("Завершение конструктора MyObject");
    }
}
```

Сравните вывод на экран такой модифицированной программы с выводом на экран исходной программы `DemonstrateCustomConstructor`:

```
Начало метода Main()
Создание локального объекта MyObject в Main():
Конструирование MyOtherObject
Конструирование MyOtherObject
Начало конструктора MyObject
(Статические члены инициализированы до конструктора)
Завершение конструктора MyObject
Нажмите <Enter> для завершения программы...
```

## Инициализация объекта без конструктора

Предположим, у вас есть небольшой класс для представления студента:

```
public class Student
{
    public string Name { get; set; }
    public string Address { get; set; }
    public double GradePointAverage { get; set; }
}
```

Объект `Student` имеет три открытых свойства, `Name`, `Address` и `GradePointAverage`, которые содержат всю основную информацию о студенте. Обычно при создании нового объекта `Student` вы должны инициализировать его свойства `Name`, `Address` и `GradePointAverage` примерно таким образом:

```
Student randal = new Student();
randal.Name = "Randal Spahr";
randal.Address = "123 Elm Street, Truth or Consequences, NM 00000";
randal.GradePointAverage = 3.51;
```

Если класс `Student` имеет конструктор, можно поступить следующим образом:

```
Student randal = new Student("Randal Spahr",
    "123 Elm Street, Truth or Consequences, NM, 00000", 3.51);
```

Однако, увы, у класса `Student` нет другого конструктора, кроме конструктора по умолчанию, автоматически создаваемого `C#`, и не принимающего никаких аргументов.



**ЗАПОМНИ!**

В `C# 3.0` и более поздних версиях можно упростить такую инициализацию при помощи кода, выглядящего подозрительно похожим на конструктор:

```
Student randal = new Student
{
    Name = "Randal Spahr",
    Address = "123 Elm Street, Truth or Consequences, NM 00000",
    GradePointAverage = 3.51
};
```

Чем отличаются эти два примера? Первый, использующий конструктор, содержит *круглые скобки*, в которые заключены две строки и одно число с плавающей точкой, разделенные запятыми. Во втором примере с применением нового синтаксиса инициализации вместо этого используются *фигурные скобки*, в которых содержатся три *присваивания*, разделенные запятыми. Этот синтаксис работает следующим образом:

```
new LatitudeLongitude
    { присваивание Latitude, присваивание Longitude };
```

Данный синтаксис инициализации объектов позволяет выполнять присваивание любому разрешающему присваивание свойству (*set*) объекта в блоке кода (в фигурных скобках). Этот блок предназначен для инициализации объекта. Заметим, что таким образом можно назначать значения только открытым свойствам, но не закрытым, а кроме того, в этом коде нельзя вызывать никакие методы объекта или выполнять какую-то иную работу.

Такой синтаксис весьма краток — одна инструкция вместо трех. Он упрощает создание инициализированных объектов, которые вы не можете инициализировать при помощи конструктора. Дает ли новый синтаксис инициализации что-либо, кроме удобства? Не многое, но удобство всегда находится в вершине списка предпочтений практикующего программиста (так же, как и краткость). Кроме того, эта возможность очень важна при работе с анонимными классами.



СОВЕТ

Пользуйтесь этой возможностью свободно, так, как вам подсказывает ваша интуиция. Если вы хотите узнать о ней побольше, поищите в справочной системе *object initializer*.

## Применение членов с кодом

Члены с кодом (*expression-bodied members*) впервые появились в C# 6.0 как средство, облегчающее определение методов и свойств. В C# 7.0 члены с кодом работают также с конструкторами, деструкторами, методами доступа к свойствам и событиям.

### Создание методов с кодом

В приведенном примере показано, как можно было создавать методы до C# 6.0:

```
public int RectArea(Rectangle rect)
{
    return rect.Height * rect.Width;
}
```



ЗАПОМНИ!

При работе с членами с кодом можно уменьшить количество строк кода до одной:

```
public int RectArea(Rectangle rect) => rect.Height * rect.Width;
```

Хотя обе версии выполняют одно и то же действие, вторая версия намного короче и ее легче написать. Компромисс заключается в том, что вторая версия может быть сложнее для понимания.

## Определение свойств с кодом

Свойства с кодом работают подобно методам: вы объявляете свойство с помощью единственной строки кода:

```
public int RectArea => _rect.Height * _rect.Width;
```

В этом примере предполагается, что у нас определен закрытый член `_rect` и что вы хотите получить значение, равное площади прямоугольника.

## Определение конструкторов и деструкторов с кодом

В C# 7.0 можно использовать тот же подход для работы с конструктором. В более ранних версиях C# можно создавать конструктор следующим образом:

```
public EmpData()  
{  
    _name = "Harvey";  
}
```

Здесь конструктор класса `EmpData` устанавливает значение закрытой переменной `_name` равным "Harvey". C# 7.0 для этого достаточно одной строки:

```
public EmpData() => _name = "Harvey";
```

Деструкторы работают в основном так же, как и конструкторы. Вместо многих строк вы можете использовать только одну.

## Определение методов доступа к свойствам с кодом

Методы доступа к свойствам также могут извлечь выгоду из членов с кодом. Вот типичный метод доступа в C# 6.0 с `get` и `set`:

```
private int _myVar;  
public MyVar {  
    get  
    {  
        return _myVar;  
    }  
    set  
    {  
        SetProperty(ref _myVar, value);  
    }  
}
```

А вот во что он превращается в C# 7.0 при использовании членов с кодом:

```
private int _myVar;
public MyVar
{
    get => _myVar;
    set => SetProperty(ref _myVar, value);
}
```

## Определение методов доступа к событиям с кодом

Как и в случае доступа к свойствам, можно создавать средства доступа к событиям, используя член с кодом. Вот что могло быть использовано в C# 6.0:

```
private EventHandler _myEvent;
public event EventHandler MyEvent
{
    add
    {
        _myEvent += value;
    }
    remove
    {
        _myEvent -= value;
    }
}
```

И вот как выглядит тот же метод доступа к событию в C# 7.0:

```
private EventHandler _myEvent;
public event EventHandler MyEvent
{
    add => _myEvent += value;
    remove => _myEvent -= value;
}
```