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

Класс должен сам отвечать за свои действия. Так же как микроволновая печь не должна вспыхнуть, объятая 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 выполнимую

.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.

Сокрытие 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 балансу 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 `Get`*`X`*`()` и `Set`*`X`*`()`, 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 GradePoint Average, 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 Sphar";
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 Sphar",
        "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 Sphar",
      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 верху 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;
}
```