Глава 11

Классы

В этой главе…

- Защита класса посредством управления доступом
- Инициализация объекта с помощью конструктора
- Определение нескольких конструкторов в одном классе
- Конструирование статических членов и членов класса

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.

Неизвестно, в каком банке храните свои сбережения вы, 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.

Управление доступом дает возможность избежать больших и малых ошибок в работе банка. Обычно программисты, привыкшие к функциональному программированию, говорят, 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, будучи переполненными 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 nAccountNumber и dBalance 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
// функция 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.dBalance += 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 nNextAccountNumber = 1000;
  private int nAccountNumber;

  // хранение баланса в виде одной переменной типа double
  private double dBalance;

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

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

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

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

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

    dBalance -= dWithdrawal;
    return dWithdrawal;
  }
```

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

В этом коде выражение `dBalance -= dWithdrawal` означает то же, что и `dBalance = dBalance - dWithdrawal`. Обычно программисты на C# стараются использовать наиболее короткую запись из возможных.

Объявляя член как `public`, вы делаете его доступным для любого кода вашей программы.

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 `dBalance` 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 существенной проблемой это может оказаться при разработке больших проектов группами программистов. Это может стать проблемой и для одного человека, поскольку ему свойственно ошибаться. Хорошо спроектированный код с правилами, выполнение которых проверяет компилятор, значительно снижает количество источников возможных ошибок.

Перед тем как идти дальше, обратите внимание, что приведенная демонстрационная программа не будет компилироваться — при такой попытке вы получите сообщение о том, что обращение к члену `DoubleBankAccount.BankAccount.dBalance` невозможно:

```
'DoubleBankAccount.BankAccount.dBalance' 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.dBalance += 10;` оказывается некорректным именно по этой причине — в силу объявления `dBalance` как `private` этот член недоступен виртуальной функции `Main()`, расположенной вне класса `BankAccount`. Замена данного выражения на `ba.Deposit(10)` решает возникшую проблему — метод `BankAccount.Deposit()` объявлен как `public`, а потому доступен для функции `Main()`.

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

В этом разделе используются определенные знания о наследовании и пространствах имен, которые будут рассмотрены в более поздних главах книги. Вы можете сейчас пропустить настоящий раздел и вернуться к нему позже, получив необходимые знания.

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 в C# называется отдельно компилируемая часть кода, представляющая собой выполнимую `.EXE`-программу либо библиотеку `.DLL`. Одно пространство имен может распространяться на несколько модулей.

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

Скрытие 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 другая функция может обращаться к этим данным, то она 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 функции. Вычислить, какие именно функции 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, когда 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.

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 функции `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 один метод для получения значения — `GetBalance()` и набор методов для получения и установки значения — `GetAccountNumber()` и `SetAccountNumber()`. Вы можете удивиться — зачем так волноваться из-за того, чтобы член `dBalance` был объявлен как `private`, и при этом предоставлять метод `GetBalance()`? На самом деле для этого имеются достаточно веские основания.

- ✓ **`GetBalance()` не дает возможности изменять член `dBalance` — 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 функции 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 нацбанка придет распоряжение хранить деньги как-то иначе, это никак не должно сказаться на вашем способе обращения с вашим счетом (по крайней мере так должно быть в цивилизованном обществе).

Пример управления доступом

Приведенная далее демонстрационная программа DoubleBankAccount указывает потенциальные изъяны программы BankAccount.

```
// 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 dDeposit = 123.454;
      Console.WriteLine("Вклад {0:C}", dDeposit);
      ba.Deposit(dDeposit);

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


      // Вот где имеется неприятность
      double dAddition = 0.002;
      Console.WriteLine("Вклад {0:C}", dAddition);
      ba.Deposit(dAddition);

      // 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 nNextAccountNumber = 1000;
    private int nAccountNumber;
```

```
// хранение баланса в виде одной переменной типа double
private double dBalance;

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

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

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

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

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

  dBalance -= dWithdrawal;
  return dWithdrawal;
}

// GetString - возвращает информацию о состоянии счета в
// виде строки
public string GetString()
```

```
    {
      string s = String.Format("#{0} = {1:C}",
                               GetAccountNumber(),
                               GetBalance());
      return s;
    }
  }
}
```

Функция `Main()` создает банковский счет и вносит на него сумму 123.454 — т.е. сумму с дробным количеством копеек. Затем функция `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. Лично я ничего не имею против округления счета до ближайшей сотни сверху, но ведь не все клиенты такие покладистые… В общем, что греха таить — в программе действительно 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 dDeposit = 123.454;
      Console.WriteLine("Вклад {0:C}", dDeposit);
      ba.Deposit(dDeposit);

      // Баланс счета
```

```
    Console.WriteLine("Счет = {0}", ba.GetString());

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

    // 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 nNextAccountNumber = 1000;
  private int nAccountNumber;

  // хранение баланса в виде одной переменной типа decimal
  private decimal mBalance;

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

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

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

```
    // Deposit - позволен любой положительный вклад
    public void Deposit(double dAmount)
    {
      if (dAmount > 0.0)
      {
        // Округление к ближайшей копейке перед внесением
        // вклада
        decimal mTemp = (decimal)dAmount;
        mTemp = Decimal.Round(mTemp, 2);

        mBalance += mTemp;
      }

    }

    // Withdraw - вы можете снять со счета любую сумму, не
    // превышающую баланс; функция возвращает реально снятую
    // сумму
    public decimal Withdraw(decimal dWithdrawal)
    {
      if (mBalance <= dWithdrawal)
      {
        dWithdrawal = mBalance;
      }

      mBalance -= dWithdrawal;
      return dWithdrawal;
    }

    // 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 функцию `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, что открытый интерфейс класса не изменился: метод `Balance()` так и возвращает значение типа `double`).

Повторюсь еще раз: приложение, использующее класс `BankAccount`, не должно изменяться при изменении класса.

В данном случае единственной функцией, которую бы пришлось изменить при непосредственном обращении к балансу, является функция `Main()`, но в реальной программе могут иметься десятки таких функций, и они могут оказаться в не меньшем количестве модулей. В анализируемом примере ни одна из этих функций не требует внесения изменений, но если бы они непосредственно обращались ко внутренним членам класса, это было бы решительно невозможно.

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 *функциями доступа* (access functions). Хотя их использование теоретически является хорошей привычкой, на практике это зачастую приводит к грустным результатам. Судите сами — чтобы увеличить на 1 член `nAccountNumber`, требуется писать следующий код:

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

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

```
public int AccountNumber        // Скобки не нужны
{
  get{return nAccountNumber;}  // Фигурные скобки и точка с
                               // запятой
  set{nAccountNumber = value;} // 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)mBalance;
  }
}
```

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 `mBalance` 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.

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

Свойства совсем не обязательно неэффективны. Компилятор C# может оптимизировать простую функцию доступа так, что она 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 должны делать и вы — даже для обращения к членам-данным класса из методов этого же класса.

Статические свойства

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

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

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

```
// Считываем свойство NextAccountNumber
int nValue = BankAccount.NextAccountNumber;
```

Побочные действия свойств

Операция `get` может выполнять дополнительную работу помимо 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 ++nNextAccountNumber;}
}
```

Это свойство увеличивает статический член класса перед тем, как вернуть результат. Однако это не слишком умная 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*.

Подобно функциям 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.

Конструирование объектов посредством конструкторов

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 заполнены “мусором”, и корректной работы от такого объекта ждать не придется.

C# решает эту проблему путем вызова инициализирующей функции автоматически. Например, в строке

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

не только выделяется память для объекта, но и выполняется его инициализация посредством вызова специальной инициализирующей функции.

Не путайте термины *класс* и *объект*. `Dog` — это класс, но собака `Scooter` — это объект класса `Dog`.

Конструкторы, предоставляемые C#

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

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

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 ошибок:

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

Однако C# предоставляет конструктор по умолчанию для объектов классов, который инициализирует члены-данные значением 0 для встроенных переменных, `false` — для логических и `null` — для ссылок. 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. Функция `Main()` создает объект класса `MyObject` и выводит начальное содержимое его членов. Вывод этой программы имеет вид

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

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".

Код, инициализирующий значения при создании, называется *конструктором*. Он "конструирует" класс в смысле инициализации его членов.

Конструктор по умолчанию

C# гарантирует, что объект начинает существование в определенном состоянии: заполненным нулями. Однако для многих классов (пожалуй, для подавляющего большинства) такое нулевое состояние не является корректным. Рассмотрим класс `BankAccount`, о котором уже шла речь ранее в этой главе.

```
public class BankAccount
{
  int nAccountNumber;
  double dBalance;
  // . . .другие члены
}
```

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

Поэтому 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 неработоспособны, хотя при этом и не будут содержать никаких ошибок. Класс не должен полагаться на внешние функции наподобие метода `InitBankAccount()`, которые должны обеспечивать корректное состояние его объектов.

Для решения данной проблемы класс предоставляет специальную функцию, автоматически вызываемую C# при создании объекта — *конструктор класса*. 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();
}
public class BankAccount
{
  // Номера банковских счетов начинаются с 1000 и
  // назначаются последовательно в возрастающем порядке
  static int nNextAccountNumber = 1000;
  // Для каждого счета поддерживаются его номер и баланс
  int nAccountNumber;
  double dBalance;
  // Конструктор BankAccount - обратите внимание на его имя
  public BankAccount() // Требуются круглые скобки, могут
                       // иметься аргументы, возвращаемый
                       // тип отсутствует
  {
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0.0;
  }
  // . . . прочие члены . . .
}
```

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

- ✓ он всегда имеет то же имя, что и сам класс;
- ✓ он не имеет возвращаемого типа, даже типа `void`;
- ✓ функция `Main()` не должна вызывать никаких дополнительных функций для инициализации объекта при его создании.

Создание объектов

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

```
// DemonstrateDefaultConstructor - демонстрация работы
// конструкторов по умолчанию; создает класс с конструктором
// и рассматриваем несколько сценариев
using System;

namespace DemonstrateDefaultConstructor
{
  // MyObject - создание класса с "многословным"
  // конструктором и внутренним объектом
  public class MyObject
  {
    // Этот член-данные является свойством класса
    static MyOtherObject staticObj = new MyOtherObject();

    // Этот член-данные является свойством объекта
    MyOtherObject dynamicObj;
    // Конструктор (с обильным выводом на экран)
    public MyObject()
    {
      Console.WriteLine("Начало конструктора MyObject");
      Console.WriteLine("(Статические члены-данные " +
                        "конструируются до вызова этого " +
                        "конструктора)");
      Console.WriteLine("Теперь динамически создаем " +
                        "нестатический член-данные:");
      dynamicObj = new MyOtherObject();
      Console.WriteLine("Завершение конструктора MyObject");
    }
  }

  // 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 функции 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 функции Main()
Создание локального объекта MyObject в Main():
Конструирование MyOtherObject
Начало конструктора MyObject
(Статические члены-данные конструируются до вызова
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 функция `Main()` выводит начальное сообщение и сообщение о предстоящем создании локального объекта `MyObject`.
2. Функция `Main()` создает объект `localObject` типа `MyObject`.
3. `MyObject` содержит статический член `staticObj` класса `MyOtherObject`. Все статические члены-данные создаются до первого запуска 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 функции `Main()`.
7. Программа выполнена.

Выполнение конструктора в отладчике

Для того чтобы выполнить рассматриваемую программу в отладчике, произведите следующие действия.

1. **Соберите программу с помощью команды меню Build⇨Build DemonstrateDefaultConstructor**.
2. **Перед тем как приступить к выполнению программы в отладчике, установите точку останова на вызове `Console.WriteLine()` в конструкторе `MyOtherObject`**.

Для установки точки останова щелкните на сером поле с левой стороны окна напротив строки, в которой хотите разместить точку останова. На рис. 11.1 показано окно отладки с точкой останова, о чем свидетельствует красная пиктограмма на серой полосе.

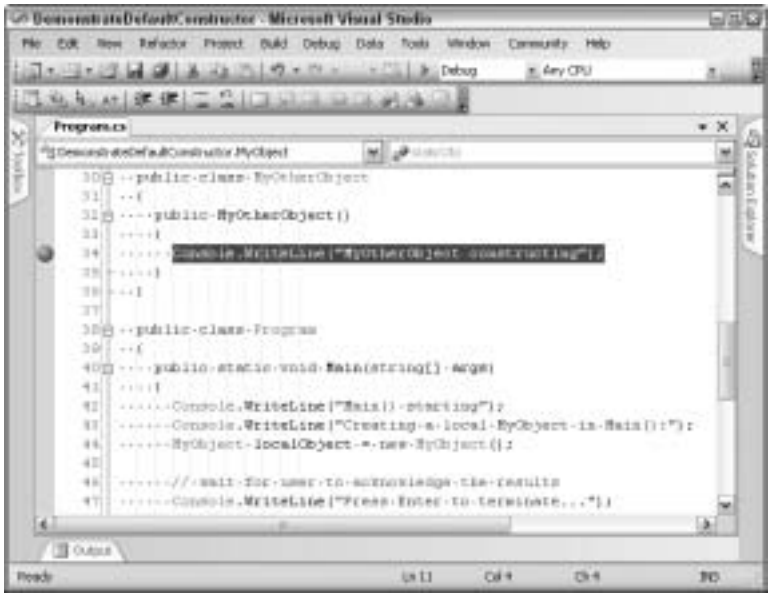

Рис. 11.1. Красная пиктограмма на серой полосе свидетельствует о наличии точки останова

3. **Воспользуйтесь командой меню Debug⇨Step Into (или нажмите клавишу <F11>)**.

   Ваши меню, полосы инструментов и окна должны немного измениться, а открывающая фигурная скобка функции `Main()` — оказаться выделенной желтым цветом фона.

4. **Нажмите клавишу <F11> еще три раза и установите курсор мыши над переменной `localObject` (без щелчка)**.

   Вы находитесь перед вызовом конструктора `MyObject`. Ваш экран должен выглядеть примерно так, как на рис. 11.2. На рисунке видно, что в настоящий мо-

мент объект `localObject` под курсором имеет значение `null`. То же показывает и окно Locals.

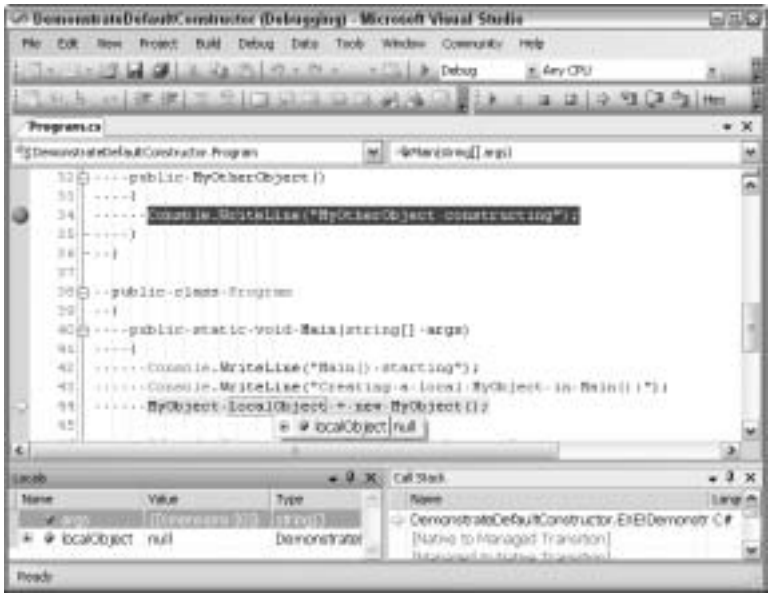

Рис. 11.2. Окно отладчика Visual Studio перед выполнением конструктора

5. **Еще раз нажмите клавишу <F11>.**

   Программа перейдет к точке останова в конструкторе `MyOtherObject`, как показано на рис. 11.3. Как мы сюда попали? Последний вызов в `Main()` приводит к запуску конструктора `MyObject`. Однако перед началом выполнения конструктора C# инициализирует статический член класса `MyObject`, который является объектом типа `MyOtherObject`, так что инициализация подразумевает вызов *его* конструктора — где и находится точка останова (без нее нельзя было бы остановить здесь отладчик, хотя сам конструктор был бы выполнен — вы бы могли судить об этом по сообщению в окне консоли).

6. **Дважды нажмите клавишу <F11>, после чего вы остановитесь на строке со статическим членом `staticObj`, как показано на рис. 11.4.**

   Это означает, что конструктор этого объекта завершил свою работу.

7. **Продолжайте нажимать клавишу <F11> для пошагового выполнения программы.**

   При первом нажатии клавиши <F11> вы остановитесь в начале конструктора `MyObject`. Обратите внимание, что вы еще раз попадете в конструктор `MyOtherObject`, но на этот раз, когда конструктор `MyObject` будет создавать нестатический член `dynamicObj`.

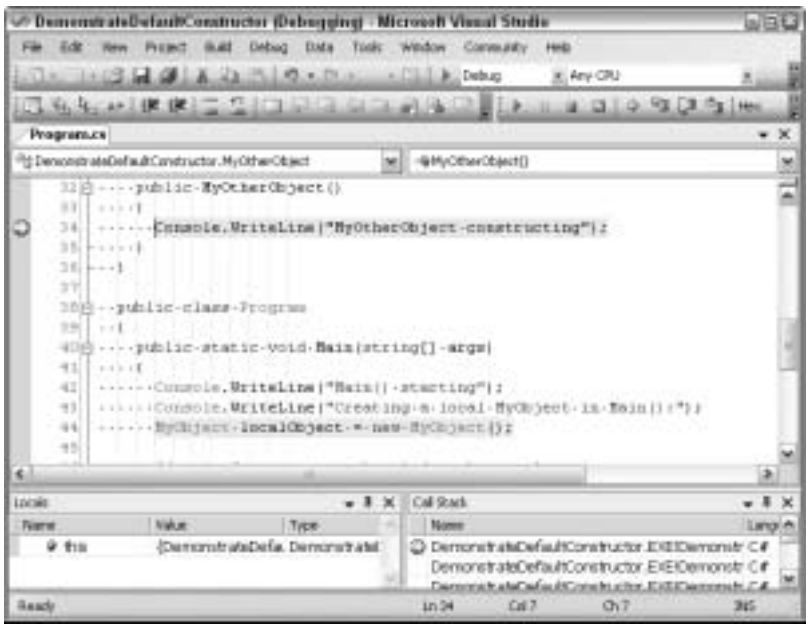

Рис. 11.3. Перед вызовом конструктора `MyObject` *управление передается конструктору* `MyOtherObject`

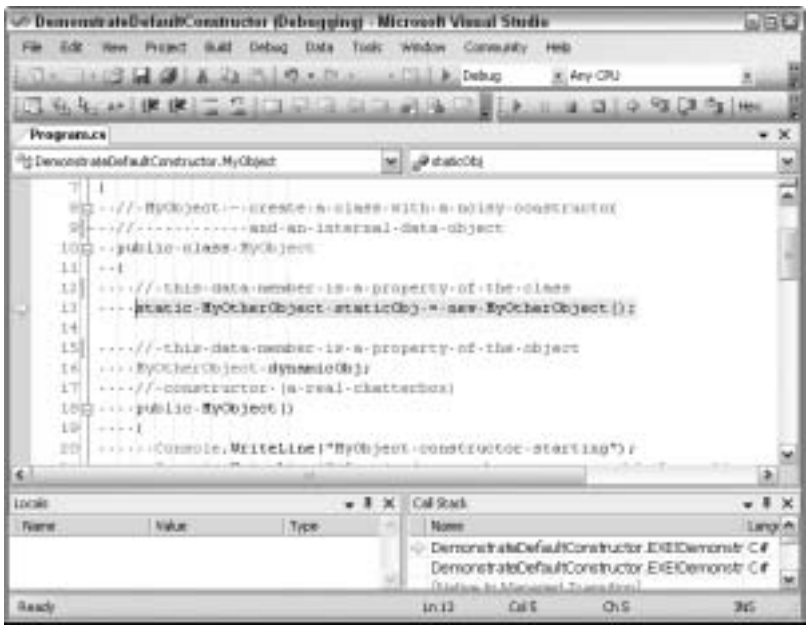

Рис. 11.4. После выполнения конструктора `MyOtherObject` *вы возвращаетесь в точку его вызова*

Непосредственная инициализация объекта — конструктор по умолчанию

Вы можете решить, что практически любой класс должен иметь конструктор по умолчанию некоторого вида, и в общем-то вы правы. Однако C# позволяет инициализировать члены-данные непосредственно, с использованием инициализаторов.

Итак, класс `BankAccount` можно записать следующим образом:

```
public class BankAccount
{
  // Номера банковских счетов начинаются с 1000 и
  // назначаются последовательно в возрастающем порядке
  static int nNextAccountNumber = 1000;
  // Для каждого счета поддерживаются его номер и баланс
  int nAccountNumber = ++nNextAccountNumber;
  double dBalance = 0.0;
  // . . . прочие члены . . .
}
```

Вот в чем состоит работа инициализаторов. Как `nAccountNumber`, так и `dBalance` 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 `dBalance` непосредственно. Но ведь `dBalance` 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 `nNextAccountNumber` инициализируется при самом первом обращении к классу `BankAccount` (как вы убедились при выполнении демонстрационной программы в отладчике), т.е. обращении к любому свойству или методу объекта, владеющему статическим членом, в том числе и конструктору. Будучи инициализирован, статический член повторно не инициализируется, сколько бы объектов вы не 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 до выполнения тела конструктора.

Конструирование с инициализаторами

Давайте в рассматривавшейся ранее программе `DemonstrateDefaultConstructor` 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
  static MyOtherObject staticObj = new MyOtherObject();
  // Этот член является свойством объекта
  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 `DemonstrateDefaultConstructor`.

```
Начало функции 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...
```

Полный текст данной программы можно найти на прилагаемом компакт-диске в каталоге `DemonstrateConstructorWithInitializer`.

Перегрузка конструкторов

Конструкторы можно перегружать так же, как и прочие методы.

Перегрузка функции обозначает определение двух функций с одним и тем же именем, но с разными типами аргументов (подробно этот вопрос рассматривается в главе 7, “Функции функций”).

Предположим, вы хотите обеспечить три способа создания объекта `BankAccount` — с нулевым балансом, как и ранее, и два варианта с некоторыми начальными значениями.

```
// BankAccountWithMultipleConstructors - банковский счет с
// разными вариантами конструкторов
using System;

namespace BankAccountWithMultipleConstructors
{
  using System;

  public class Program
  {
    public static void Main(string[] args)
    {
      // Создание банковских счетов с корректными начальными
      // значениями
      BankAccount ba1 = new BankAccount();
      Console.WriteLine(ba1.GetString());
```

```
    BankAccount ba2 = new BankAccount(100);
    Console.WriteLine(ba2.GetString());

    BankAccount ba3 = new BankAccount(1234, 200);
    Console.WriteLine(ba3.GetString());

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

// BankAccount - простейший банковский счет
public class BankAccount
{
  // Первый номер счета – 1000; номера счетов
  // назначается последовательно
  static int nNextAccountNumber = 1000;

  // Номер счета и его баланс
  int nAccountNumber;
  double dBalance;

  // Несколько конструкторов - в зависимости от ваших
  // потребностей
  public BankAccount() // Автоматического конструктора нет
  {
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0.0;
  }

  public BankAccount(double dInitialBalance)
  {
    // Повторение части кода из конструктора по умолчанию
    nAccountNumber = ++nNextAccountNumber;

    // Теперь – код, специфичный для данного конструктора
    // Начинаем с переданного баланса (если он
    // положительный)
    if (dInitialBalance < 0)
    {
      dInitialBalance = 0;
    }
    dBalance = dInitialBalance;
  }

  public BankAccount(int nInitialAccountNumber,
                     double dInitialBalance)
  {
    // Игнорируем отрицательный номер счета
    if (nInitialAccountNumber <= 0)
    {
```

```
        nInitialAccountNumber = ++nNextAccountNumber;
      }
      nAccountNumber = nInitialAccountNumber;

      // Начинаем с переданного баланса (если он
      // положительный)
      if (dInitialBalance < 0)
      {
        dInitialBalance = 0;
      }
      dBalance = dInitialBalance;
    }

    public string GetString()
    {
      return String.Format("#{0} = {1:N}",
                           nAccountNumber, dBalance);
    }
  }
}
```

Как только вы определили собственный конструктор — не имеет значения, какого именно типа, C# больше не создает конструктор по умолчанию для вашего класса. Поэтому нужно самим определить конструктор без параметров в этой демонстрационной программе.

В приведенной выше демонстрационной программе `BankAccountWithMultipleConstructors` имеются три конструктора:

- ✓ первый назначает номер счета и нулевой баланс;
- ✓ второй назначает номер счета и инициализирует баланс переданным положительным значением (отрицательные значения игнорируются);
- ✓ третий позволяет пользователю самостоятельно определить номер счета и положительный начальный баланс.

Функция `Main()` создает различные банковские счета с использованием каждого из трех конструкторов и выводит информацию о созданных объектах. Вывод этой программы на экран имеет следующий вид:

```
#1001 = 0.00
#1002 = 100.00
#1234 = 200.00
Нажмите <Enter> для завершения программы...
```

В реальных классах требуется выполнять более строгую проверку входных параметров конструктора, чтобы гарантировать их корректность.

Конструкторы различаются между собой по тем же правилам, что и перегруженные функции. Первый объект, конструируемый в функции `Main()`, `ba1`, создается без аргументов, так что для него вызывается первый конструктор `BankAccount()`, не получающий аргументов (он все еще именуется конструктором по умолчанию, хотя и не создается C# автоматически). Соответственно, этот счет получает номер по умолчанию и нулевой ба-

ланс. Второй счет ba2 использует конструктор BankAccount(double) и, соответственно, получает очередной свободный номер, но его начальный баланс равен переданному значению 100. Третий счет ba3 получает при конструировании два параметра, так что для него вызывается конструктор BankAccount(int,double), и счету присваивается индивидуальный номер и начальный баланс.

Устранение дублирования конструкторов

Подобно типичному сценарию мыльной оперы, три конструктора BankAccount во многом дублируют друг друга. Как нетрудно понять, в реальных задачах дела обстоят еще хуже, так как и членов, и конструкторов в реальных классах может быть существенно больше. Кроме того, сложные проверки корректности входных параметров могут перекрываться и служить источником ошибок — в частности, при их синхронизации из-за изменения бизнес-правил.

Хотелось бы иметь возможность вызывать один конструктор из другого, но это нереально, так как конструкторы не функции, и они не вызываются. Однако можно создать некоторую функцию, выполняющую инициализацию, и вызывать ее в конструкторах, что и показано в демонстрационной программе BankAccountContructorsAndFunction.

```
// BankAccountContructorsAndFunction - банковский счет с
// несколькими конструкторами
using System;

namespace BankAccountContructorsAndFunction
{
  using System;

  public class Program
  {
    public static void Main(string[] args)
    {
      // Создание банковского счета с корректными начальными
      // значениями
      BankAccount ba1 = new BankAccount();
      Console.WriteLine(ba1.GetString());

      BankAccount ba2 = new BankAccount(100);
      Console.WriteLine(ba2.GetString());

      BankAccount ba3 = new BankAccount(1234, 200);
      Console.WriteLine(ba3.GetString());

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

```
  // BankAccount - банковский счет
  public class BankAccount
  {
    // Первый номер счета — 1000; номера счетов
    // назначается последовательно
    static int nNextAccountNumber = 1000;

    // Номер счета и его баланс
    int nAccountNumber;
    double dBalance;

    // Размещаем весь инициализирующий код в отдельной
    // функции, вызываемой из конструкторов
    public BankAccount() // Автоматического конструктора нет
    {
      Init(++nNextAccountNumber, 0.0);
    }

    public BankAccount(double dInitialBalance)
    {
      Init(++nNextAccountNumber, dInitialBalance);
    }

    // Конструктор с наибольшим количеством аргументов
    // выполняет всю работу
    public BankAccount(int    nInitialAccountNumber,
                       double dInitialBalance)
    {
      // На самом деле тут надо проверить, чтобы значение
      // nInitialAccountNumber (а) не совпадало с уже
      // назначенными номерами счетов и (б) было не меньше
      // 1000
      Init(nInitialAccountNumber, dInitialBalance);
    }

    private void Init(int    nInitialAccountNumber,
                      double dInitialBalance)
    {
      nAccountNumber = nInitialAccountNumber;

      // Используем переданный баланс (если он положителен)
      if (dInitialBalance < 0)
      {
        dInitialBalance = 0;
      }
      dBalance = dInitialBalance;
    }

    public string GetString()
    {
      return String.Format("#{0} = {1:N}",
                           nAccountNumber, dBalance);
    }
  }
}
```

Здесь метод `Init()` выполняет всю работу, связанную с конструированием. Однако по ряду причин такой подход недостаточно "кошерный" — не в последнюю очередь из-за того, что при этом вызывается метод объекта, который еще не полностью построен, а это очень опасная игра!

К счастью, такой подход не является необходимым. Один конструктор может обращаться к другому с использованием ключевого слова `this` следующим образом:

```
// BankAccountContructorsAndThis - банковский счет с
// несколькими конструкторами
using System;

namespace BankAccountContructorsAndFunction
{
  using System;

  public class Program
  {
    public static void Main(string[] args)
    {
      // Создание банковского счета с корректными начальными
      // значениями
      BankAccount ba1 = new BankAccount();
      Console.WriteLine(ba1.GetString());

      BankAccount ba2 = new BankAccount(100);
      Console.WriteLine(ba2.GetString());

      BankAccount ba3 = new BankAccount(1234, 200);
      Console.WriteLine(ba3.GetString());

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

  // BankAccount - банковский счет
  public class BankAccount
  {
    // Первый номер счета – 1000; номера счетов
    // назначается последовательно
    static int nNextAccountNumber = 1000;

    // Номер счета и его баланс
    int nAccountNumber;
    double dBalance;

    // Вызываем конструктор с наибольшим количеством
    // аргументов, передавая значения по умолчанию для
    // отсутствующих параметров
```

```
    public BankAccount() : this(0, 0) {}

    public BankAccount(double dInitialBalance)
        :this(0, dInitialBalance) {}

    // Конструктор с наибольшим количеством аргументов
    // выполняет всю работу
    public BankAccount(int nInitialAccountNumber,
                       double dInitialBalance)
    {
      // Игнорируем отрицательные номера счетов; нулевое
      // значение означает, что мы хотим использовать
      // очередное свободное значение номера счета
      if (nInitialAccountNumber <= 0)
      {
        nInitialAccountNumber = ++nNextAccountNumber;
      }
      nAccountNumber = nInitialAccountNumber;

      // Используем переданный баланс (если он положителен)
      if (dInitialBalance < 0)
      {
        dInitialBalance = 0;
      }
      dBalance = dInitialBalance;
    }

    public string GetString()
    {
      return String.Format("#{0} = {1:N}",
                           nAccountNumber, dBalance);
    }
  }
}
```

В этой версии класса `BankAccount` имеются те же три варианта конструкторов, что и в предыдущих демонстрационных программах. Однако вместо повторения некоторых проверок в каждом конструкторе более простые конструкторы вызывают наиболее сложный и гибкий конструктор с использованием значений по умолчанию, а в нем и выполняются все необходимые проверки. Наличие функции `Init()` становится ненужным.

Создание объекта с использованием конструктора по умолчанию включает вызов конструктора `BankAccount()`:

```
BankAccount ba1 = new BankAccount(); // Параметров нет
```

Конструктор `BankAccount()` тут же передает управление конструктору `BankAccount(int,double)`, передавая ему значения по умолчанию 0 и 0.0:

```
public BankAccount() : this(0, 0) {}
```

Поскольку тело конструктора пустое, весь конструктор можно смело записывать в одну строку.

После выполнения основного конструктора с двумя параметрами управление возвращается конструктору по умолчанию, тело которого в данном случае совершенно пустое.

Создание банковского счета с ненулевым балансом и номером счета по умолчанию осуществляется следующим образом:

```
public BankAccount(double d) : this(0, d) {}
```

Фокусы с объектами

Невозможно создать объект без конструктора какого-либо вида. Если вы определите собственный конструктор, C# будет работать только с ним. Объединяя эти два факта, можно создать класс, который может быть инстанцирован только локально.

Например, создать объект `BankAccount`, если конструктор объявлен как `internal` (показан полужирным шрифтом в приведенном далее исходном тексте), могут только методы, определенные в том же модуле, что и `BankAccount`.

```
// BankAccount - моделирование простейшего банковского счета
public class BankAccount
{
  // Номера банковских счетов начинаются с 1000 и
  // назначаются последовательно в возрастающем порядке
  static int nNextAccountNumber = 1000;
  // Для каждого счета поддерживаются его номер и баланс
  int nAccountNumber;
  double dBalance;

  internal BankAccount() // Конструктор
  {
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0;
  }
  public string GetString()
  {
    return String.Format("#{0} = {1:N}",
                         nAccountNumber, dBalance);
  }
}
```