МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ
ОДЕССКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ
имени И.И.МЕЧНИКОВА
Институт математики, экономики и механики
Кафедра математического обеспечения компьютерных систем

В. Г. ПЕНКО, Е. А. ПЕНКО

Программное обеспечение ЭВМ.
Часть 1

Методическое пособие для студентов отделения прикладной математики и факультета информационных технологий 2—3 курсов дневного/заочного отделений

ОДЕССКИЙ НАЦИОНАЛЬНЫЙ
УНИВЕРСИТЕТ
имени И.И.МЕЧНИКОВА
2010

Программное обеспечение ЭВМ. Часть 1

Методическое пособие содержит описание языка программирования C#, рассмотрены основные конструкции и методики объектно-ориентированного программирования.

В пособии делается акцент на практическое освоение современных средств объектно-ориентированного программирования в контексте использовании среды программирования .NET.

Предназначено для студентов, обучающихся по направлению «Прикладная математика» и всех тех, кто стремится самостоятельно научиться программировать на C#.

Авторы:

В.Г. Пенко, канд. техн. наук, доцент кафедры математического обеспечения компьютерных систем

Е.А. Пенко, ассистент кафедры математического обеспечения компьютерных систем

Рецензенты:

Т.И. Петрушина, кандидат физ.–мат. наук, доцент

Ю.М. Крапивный, кандидат физ.–мат. наук, доцент

Рекомендовано к изданию Учёным советом
Института математики, экономики и механики
ОНУ имени И.И. Мечникова.
Протокол № 1 от 9 октября 2009 г.

СОДЕРЖАНИЕ

Введение в программирование на C# в .NET

Что такое .NET и зачем она нужна?

Многие книги, тематически относящиеся к .NET, начинаются с довольно пространной главы, подробно объясняющей роль .NET. Нам кажется, что подобный подход не совсем удачен. В данном курсе будем придерживаться другой стратегии – компоненты и свойства .NET будут описываться по мере того, как они будут проявляться в процессе создания программных приложений.

Пока остановимся на таком рабочем определении – среда .NET для программиста играет примерно ту же роль, что операционная система для пользователя, то есть приподнимает уровень средств программирования, делая их концепции более близкими к естественным (с точки зрения программиста) и, как следствие, более эффективными в процессе использования.

Благодаря такой роли, .NET прекрасно справляется с задачей обеспечения общего фундамента сразу для нескольких языков программирования. Среди них наиболее актуальными являются C++, Visual Basic, J# и C# (Си шарп). Именно C# будет использоваться нами далее для демонстрации возможностей .NET.

Следует четко отделять полезную функциональность, предоставляемую средой .NET и системой программирования. Существует несколько систем программирования на базе .NET – MS Visual Studio, Sharp Developer и др. Система программирования – это еще один слой, обеспечивающий удобство программирования. И хотя упомянутые системы программирования базируются на одинаковой платформе .NET, они все же отличаются количеством и уровнем услуг. К примеру, в разных системах с разной степенью могут быть реализованы средства автозавершения кода.

Первая программа на C# и основные приемы работы в системе MS Visual Studio

После запуска MS Visual Studio (далее – VS) следует выполнить команду File →New →Project. Далее в диалоговом окне выбрать один из доступных языков программирования (обычно – C#, C++, Visual Basic и J#). Наш курс построен на основе языка C#. После выбора языка C# следует выбрать одну из разновидностей (template – шаблонов) приложения. Шаблон Console Application мы будем использовать для создания учебных приложений, не использующих стандартный оконный интерфейс взаимодействия с пользователем (за это отвечает шаблон Windows Application). Далее в том же окне напечатайте придуманное имя проекта

(вместо стандартного ConsoleApplication1 или WindowsApplication1) и укажите папку – место расположения проекта в файловой системе Вашего компьютера.

Допустим, Вы назвали проект FirstApp и выбрали шаблон Console Application. Что такое проект? Разве нельзя просто написать программу? В современных системах программирования даже простейшая программа должна храниться в исходном файле, являющемся частью проекта. В более сложных ситуациях кроме текста на языке программирования, программа будет использовать некоторые дополнительные ресурсы, например, рисунок для значка приложения. Удобно, если все такие ресурсы и файлы с исходными текстами будут храниться в одном месте. Это место и называется проектом. В нашем случае VS создала проект `FirstApp` с одним исходным файлом Program.cs (cs – стандартное расширение для языка C#). VS даже сгенерировала некоторый текст программы:

```
01.   using System;
02.   using System.Collections.Generic;
03.   using System.Text;
04.   namespace FirstApp
05.   {
06.      class Program
07.      {
08.         static void Main(string[] args)
09.         {
10.         }
11.      }
12.   }
```

Номера строк приведены здесь для удобства – в VS эти номера не видны. Эта «непонятная» программа даже работает, если Вы выполните команду Start Debugging (F5 или кнопка ▶). После запуска программа на мгновение показывает черное консольное окно и «успешно» завершает работу.

Завершите работу VS (не забыв сохранить изменения) и исследуйте файловую систему – в папке, указанной вами для хранения проекта была создана папка `FirstApp` с таким содержимым:

Несколько неожиданно, что в папке FirstApp имеется еще одна вложенная папка FirstApp. Сам исходный файл Program.cs находится именно

во вложенной папке. Дело в том, что при создании нового проекта VS автоматически создает еще и так называемое решение (solution). Решение может содержать несколько проектов. Это удобно, когда разработчик одновременно занимается несколькими сходными проектами. Поэтому папка FirstApp окружена еще одним группирующим слоем – папкой FirstApp для решения. В этой папке имеется специальный файл решения с расширением sln. Щелкните по файлу FirstApp.sln дважды и Вы запустите VS с уже загруженным решением FirstApp. Позже Вы узнаете, как добавлять в решение проект.

Чтобы программа стала более содержательной, после строки 08 добавьте следующие строки:

```
09.   string a, b, c; // описание строковых переменных
10.   a = Console.ReadLine(); //ввод двух строк
11.   b = Console.ReadLine(); //с клавиатуры
12.   c = a + b;              //конкатенация строк
13.   Console.WriteLine(c);   //вывод строки на экран
```

Вы наверняка почувствовали, что язык C# весьма похож на язык C++. Но уже сейчас будьте внимательны – имеются также и существенные отличия.

Строка 09 содержит почти обычный оператор описания трех переменных и комментарий в конце строки. Здесь важно заметить, что переменные a, b и c относятся к встроенному в C# типу string – полноценному типу данных, поддерживающему работу со строками. Может быть кто-то помнит, как «занимательно» было писать программы, работающие со строками в старом добром языке C. Помните, что строки можно было моделировать даже двумя способами (как массив символов или адресуемая указателем область памяти). В конце каждой строки следовало поместить символ-терминатор ‘\0’. А еще Вам предоставляли много-много функций для работы со строками. Только в разных системах программирования наборы этих функций «немножко» отличались и описывались в разных заголовочных файлах.

Используя тип string в C#, Вы сможете «почувствовать разницу»,. Ввести содержимое строки с клавиатуры теперь можно с помощью конструкции `Console.ReadLine()` – это похоже на вызов функции, возвращающей значение, которое копируется в переменную с помощью оператора присваивания. Оператор в строке 12 интуитивно понятен (для склеивания строк в C++ предлагалось использовать неочевидную функцию `strcat)`. Наконец, вывод на экран происходит с помощью строки 13.

Очень важно понять, как выполняется ввод и вывод. Здесь использован тяжеловесный, по сравнению с объектами `cin` и `cout`, синтаксис. В обоих случаях используется один класс `Console`. Он

обеспечивает несколько функций (методов) для управления консолью. Среди них – метод `ReadLine` для ввода с клавиатуры и методы `Write` и `WriteLine` для вывода на экран. Как и C++, язык C# чувствителен к регистру – «опечатка» `Writeline` приведет к ошибке на этапе компиляции.

Метод `WriteLine` после вывода своего параметра на экран автоматически переводит курсор в начало новой строки. Метод `Write` оставляет курсор в текущей строке. По-прежнему можно использовать и «старый» способ перехода на новую строку – управляющий символ '\n'. Например,

```
Console.Write("Hello,\nmy friend!\n");
```

выводит на экран две строки и переводит курсор в начало третьей.

Метод `ReadLine` позволяет пользователю ввести строку символов. После нажатия клавиши Enter введенная строка становится возвращаемым значением метода `ReadLine`.

Замечание: не используйте похожий метод `Read`. Почему? Ищите ответ в справочной системе VS.

Только что мы использовали очень важное понятие - класс. Сейчас его можно упрощенно (!) определить как программную конструкцию, обеспечивающую ряд сходных функций. Если в VS Вы напечатаете в тексте программы название класса с последующей точкой, то VS «сообразит», что Вы хотите воспользоваться одним из методов класса и покажет «умный» список со всеми элементами класса (такие списки действительно называются IntelliSense– здравый смысл).

Для вызова метода класса, следует использовать следующий синтаксис:

```
<имя класса>.<имя метода>()
```

Скобки после имени метода могут содержать один или несколько параметров. Если метод возвращает значение, его часто используют в правой части оператора присваивания или внутри выражения.

Класс `Console`, как и многие другие полезные классы входит в состав среды .NET. Эту компоненту .NET называют Basic Class Library (BCL). Поражает количество классов – более 2000. Если каждый из классов предоставляет по 10 методов, то количество методов просто ошеломляет. Поэтому классы хотя бы частично решают проблему группировки большого количества методов. Для дальнейшей классификации используются так называемые пространства имен (namespaces). Например, в наиболее популярном пространстве имен `System` имеется 113 классов. Их было бы еще больше, если само пространство имен `System` не состояло из нескольких вложенных пространств имен. Например, пространство имен System.IO содержит 34 класса, имеющих отношение к файловому вводу/выводу и управлению файловой системой.

Таким образом, система классов и пространств имен напоминает файловую систему, где роль файлов играют классы, а роль папок – пространства имен.

Из соображений эффективности приложение не может использовать любые классы BCL. Если программист хочет использовать в программе некоторый класс, он должен написать в начале исходного текста программы директиву using, указывающую пространство имен, содержащее этот файл. Как видите, в простейшей программе VS сгенерировала подключение трех пространств имен:

```
using System;
using System.Collections.Generic;
using System.Text;
```

Итак, определилось одно из многочисленных свойств .NET – она предоставляет обширнейший набор полезных функций в виде методов классов BCL.

Модернизируем нашу программу, чтобы она работала не со строками, а с целыми числами. На первый взгляд для этого нужно только заменить описание переменных a, b и c:

```
int a, b, c; // описание целых переменных
```

Действительно, в языке C# сохранилась возможность описывать таким образом переменные для представления целых чисел. В .NET им отводится 32 бита в памяти. Однако при попытке применить метод `Console.WriteLine`, компилятор выдает следующую ошибку:

Cannot implicitly convert type 'string' to 'int'

В документации сказано, что метод `ReadLine` возвращает строку символов (тип `string`). Однако неявного преобразования (implicit conversion) при присваивании целой переменной C# делать не собирается. Возникает необходимость явного преобразования. Не следует путать приведение и преобразование типов. Приведение типов не требует выполнения преобразования информации и применяется, когда программист хочет сказать, что в некоторой области памяти или переменной уже находится значение некоторого типа. Если в действительности в памяти не находится соответствующее значение, то при выполнении программы возникнет ошибка. Преобразование типа – это процедура преобразования информации. Очевидно, что в нашем случае требуется преобразование. Где искать соответствующую возможность. Если Вы уже прониклись духом объектно-ориентированных систем программирования, то правильно ответите – нужно искать метод некоторого класса. Наша проблема может быть решена с помощью класса – специалиста по подобным преобразованием. У него и название соответствующее – `Convert`.

Теперь для получения значений переменных a и b нужно использовать такие операторы:

```
a = Convert.ToInt32(Console.ReadLine());
b = Convert.ToInt32(Console.ReadLine());
```

Итак, метод `ToInt32` осуществляет преобразование строки (возвращаемое значение `Console.ReadLine() )` в целое число. Здесь можно отметить две особенности. Во-первых, целый тип данных, на самом деле в C# реализован классом `Int32` (отсюда название метода `ToInt32`). Возможность описывать переменные как `int` осталась только для синтаксической совместимости с C++. Во-вторых, следует отдать должное универсальности класса `Convert`. Он умеет преобразовывать значения любых 16 встроенных типов в значения любого из этих встроенных типов (т.е. несколько сотен вариантов преобразований). Для этого у класса `Convert` имеются 16 методов: `ToInt32`, `ToDouble` и т.д. Более того, каждый из этих методов имеет 16 вариантов, отличающихся друг от друга типом параметров – это явление называется **перегрузкой метода**.

Структура программы

Сразу предупредим – в данной главе нет исчерпывающего изложения соответствующей информации. Целью главы является выделение основных структурных свойств программы на языке C#. Постепенно Вы будете узнавать другие подробности на эту тему.

Напомним, что главной структурной единицей программы на языке C++ была функция, а программа состояла из множества таких функций, среди которых обязательно должна быть функция `main`. Кроме этого на верхнем уровне можно было описывать глобальные переменные и собственные типы данных, такие как `struct`.

Идея создания программистом собственных типов данных в языке C# приобрела центральное место (как и в других объектно-ориентированных языках). По-прежнему это можно делать с помощью структурных типов `struct`. Однако еще более полноценную реализацию понятия тип удалось реализовать с помощью концепции классов.

Такие классы, как `Console` и `Convert` называются встроенными только по той причине, что они написаны другими программистами и включены в стандартный набор классов. Однако программист может создавать свои собственные классы, обеспечивая их необходимыми ему функциями-методами.

В языке C# самостоятельных функций вообще нет – есть только методы классов, а вся программа – это просто несколько классов, описанных один за другим. Кроме того, пропали глобальные переменные – как

убедились разработчики, глобальные переменные привносят в программу много опасных проблем.

Есть еще один важный вопрос: если структура программы однородна и представляет множество классов, то с какого места начинается выполнение программы. Для этого в одном (и только в одном) из классов обязательно доложен быть определен метод `Main` (с большой буквы!). Таким образом, ошибкой будет и отсутствие метода `Main` и наличие нескольких методов `Main`.

Наша первая программа удовлетворяет перечисленным требованиям. Вся она состоит из единственного класса `Program`. В этом классе содержится только один метод, и он называется `Main` – как говорится минимальный джентльменский набор.

Ответ на вопрос «что означает слово `static` в заголовке метода `Main`» вы узнаете позже, а на вопрос «зачем нужны параметры в методе `Main`» мы в нашем кратком курсе отвечать не будем – нельзя объять … сами знаете что.

Еще вопросы есть? По-крайней мере еще один важный вопрос должен остаться – что означает конструкция `namespace` в нашей программе.

Собственные пространства имен

Мы уже говорили, что пространства имен – это способ группировки классов. А если речь идет о классах, разработанных самим программистом? Здесь также используются пространства имен. Достаточно окружить несколько описаний классов конструкцией `namespace`, и они уже образуют новое пространство имен:

```
namespace AAA
{ <описание класса>
. . .
  <описание класса>
}
```

Пространство имен может группировать классы, находящиеся в разных исходных файлах. И, наоборот, в одном исходном файле могут содержаться классы из разных пространств имен. Ниже соответствующий пример.

Файл x.cs	Файл y.cs

namespace AAA
{ class A {. . .}
  class B {. . .}
}
namespace BBB
	namespace BBB
	{ class E {. . .}
	  class F {. . .}
	}
	namespace AAA

`{ class C {. . .}`
`  class D {. . .}`
`}`
	`{ class G {. . .}`
	`  class H {. . .}`
	`}`

Особенности языка C#

Ранее рассмотренные особенности программирования на C# не относились к самому языку, а лишь касались множества доступных для использования методов классов. В данной главе обратим внимание на некоторые особенности в наиболее стабильной части языка – в стандартных операторах. Большинство этих особенностей является результатом очищения языка C++ от своего тяжелого наследия в лице сравнительно низкоуровневого предшественника C.

Полноценный логический тип данных

Помните, как замысловато был представлен логический тип данных в C – числовое значение 0 воспринималось как «ложь», а любое другое – как «истина». Так что цикл `while(5){. . .}` хотя и выглядел странновато, оказывался бесконечным циклом. Кроме того, в C операция присваивания имела два результата – побочный (сам факт копирования значения в правую часть) и основной – значение правой части после присваивания. Это позволяло выполнять цепочечные присваивания (a=b=c;). В силу этих двух особенностей многие невнимательные программисты (а кто же всегда будет внимателен?) допускали маленькую опечатку – вместо операции сравнения == в условных выражения использовали операцию присваивания =). Хуже всего, что с точки зрения компилятора эта конструкция воспринималась правильно. В результате цикл `while(a=5){. . .}` также становился бесконечным циклом.

В языке C# никакого хитроумного моделирования логических значений нет – вместо этого имеется полноценный логический тип `bool`, на самом деле реализованный классом `Boolean` (помните `int` и `Int32`?).

В результате компилятор обнаруживает ошибку в условном выражении `a=1` и сообщает, что не может неявно преобразовать целую величину в логическую.

Оператор switch

Отметим две особенности, отличающие оператор switch в языках C# и C++:

1. C# не поддерживает “провала” передачи управления, возникающего при отсутствии оператора `break`.

   Фрагмент программы

```
int a; a=2;
switch (a)
{ case 1: cout<<”Один”;
  case 2: cout<<”Два”;
  case 3: cout<<”Три”;
  case 4: cout<<”Четыре”;
}
```

приводил к выводу на экран строки «ДваТриЧетыре».

В C# аналогичный оператор будет считаться компилятором ошибочным – C# требует в конце каждого исполняемого блока указывать оператор `break`, либо `goto`, либо `return`. Таким образом, можно написать следующее:

```
int a; a=2;
switch (a)
{ case 1: Console.Write(”Один”); break;
  case 2: Console.Write(”Два”); break;
  case 3: Console.Write(”Три”); break;
  case 4: Console.Write(”Четыре”); break;
}
```

В результате на экране появится только слово «Два»

2. В качестве выражения и констант, отмечающих варианты оператора `switch` можно использовать строковые данные (тип `string`)

```
string a; a=”Два”;
switch (a)
{ case ”Один”: Console.Write(1); break;
  case ”Два”: Console.Write(2); break;
  case ”Три”: Console.Write(3); break;
  case ”Четыре”: Console.Write(4); break;
}
```

Такой фрагмент выводит на экран число 2.

Основные классы

Ниже рассмотрим использование наиболее популярных классов платформы .NET

Класс Console

Как вы уже успели заметить, этот класс используется для организации взаимодействия пользователя с компьютером с помощью двух устройств: клавиатура (для ввода информации) и монитор (для вывода). В предыдущих примерах были использованы методы `ReadLine` и `WriteLine`. Отметим ещё одну полезную особенность метода `WriteLine`.

Допустим, нужно вывести на экран значения трёх переменных x, y и z. это можно сделать следующим образом:

```
Console.WriteLine("x="+x+", y="+y+", z="+z);
```

При вводе такого оператора приходится внимательно следить за соблюдением порядка множества элементов, составляющих параметр метода. Намного удобнее этой же цели добиться иначе:

```
Console.WriteLine("x={0}, y={1}, z={2}",x,y,z);
```

В этом случае метод `WriteLine` имеет несколько параметров. Первый обязательно является строкой, определяющей структуру выводимой информации. В этой строке присутствуют специальные элементы (`{0}{1}{2}`), вместо которых выводятся значения соответствующих других параметров (в нашем примере x, y и z).

Кроме `ReadLine` и `WriteLine` класс `Console` содержит дополнительные методы для управления клавиатурой и выводом информации на монитор. Например, с помощью метода `SetCursorPosition` можно управлять позицией курсора.

Класс Convert

Мы уже знаем о существовании метода `ToInt32`, осуществляющего преобразование в целое число. Кроме того, в этом классе имеется ещё ряд методов преобразования: `ToBoolean`, `ToByte`, `ToChar`, `ToDateTime`, `ToDecimal`, `ToDouble`, `ToSingle`, `ToString`. Этот (неполный!) перечень методов позволяет сделать следующие выводы:

1. Система основных типов языка в основном унаследована от языка C++, однако, в ней появились и новшества.

   Как уже говорилось, в C# имеется полноценный логический тип данных `Boolean`.

   К числу основных типов в C# добавился тип `DateTime`, позволяющий оперировать календарными датами и временем.

   Тип `Decimal` это разновидность вещественного типа данных используемого, когда важно избежать накопления погрешностей вычислений.

   Два основных вещественных типа языка C++ (`float` и `double`) в C# «поменялись местами» - стандартным теперь является `Double`. Это означает, что вещественные константы, записываемые обычным образом, относятся к типу `Double`. Поэтому оператор `a = 0.5` будет признан компилятором ошибочным (несовместимость типов в присваивании), если переменная `a` имеет тип `float`.

   Преобразованием в тип `float` занимается метод `ToSingle`.

2. Каждый из методов преобразования является перегруженным. Это позволяет применять метод с одним и тем же именем для преобразования разнотипной исходной информации. Например:

```
Convert.ToInt32(3.2);
Convert.ToInt32(false);
Convert.ToInt32("Hello");
```

В связи с этим полезно будет выполнть следующее задание: написать программу, проверяющую результаты применения методов преобразования к различным типам данных. Выявить комбинации типов, приводящие к ошибкам и дать этому объяснение.

Строковый тип данных string (класс String)

Работу со строками символов в .NET очень удобно осуществлять с помощью типа данных `String` из пространства имен `System`. Переменные этого типа можно описывать также с помощью ключевого слова `string`.

Переменная строкового типа может получить свое значение несколькими способами:

1. Присваиванием строковой константы или строкового выражения:

```
String name;
name="John";
name="Mr. " + name; //для строк + действует как конкатенация
```

2. Путем ввода с клавиатуры

```
name=Console.Readline();
```

Сравнивать строки можно операциями == (равно) и != (не равно). При этом проверяется точное совпадение содержимого этих строк. Однако применять для сравнения операции <, >, >= и <= нельзя. Для таких сравнений существует специальный метод `Compare` в классе `String`.

В C# сохранилась возможность работать со строкой как с массивом символов.

В следующем фрагменте программы проиллюстрированы некоторые из перечисленных возможностей:

```
string s1,s2;
s1 = Console.ReadLine().ToLower();
s2 = "";    //строка может быть пустой
for (int i = 0; i < s1.Length; i++) s2 += s1[s1.Length-i-1];
Console.WriteLine(s2);
```

Здесь сначала описываются две строковые переменные `s1` и `s2`. Далее с помощью метода `ToLower` вводимая с клавиатуры строка переводится в нижний регистр и результат присваивается переменной `s1`. Обратите

внимание на цепочку вызовов методов `Console.ReadLine().ToLower()`. Сначала `ReadLine` возвращает строку, а затем `ToLower` переводит ее в нижний регистр.

Далее выполняется цикл, переворачивающий содержимое строки `s1`. Перевернутое содержимое оказывается в строке `s2`. Обратите внимание на то, что для определения длины строки используется `Length`. Вы не видите спика параметров в круглых скобках после `Length`. Это значит, что `Length` не является методом, а свойством класса. Позже мы обсудим свойства подробнее.

Обращение к символам строки с помощью индексации возможно только для чтения. Попытка изменить строку с помощью индексации приводит к ошибке компиляции:

```
s[0]='!';  //Ошибка
```

Класс `String` предлагает обширный набор полезных методов. В следующей таблице указаны некоторые из них.

Таблица членов класса `String`

Член	**Описание**
`Length`	Свойство, возвращающее длину текущей строки
`Compare()` и `CompareTo()`	Методы, сравнивающие два заданных объекта `String` и возвращающие целое число, которое показывает их связь друг с другом в порядке сортировки
`Contains()`	Метод, применяемый для выяснения того, содержит ли текущий строковый объект данную строку
`Copy()`	Метод, создающий новый экземпляр `String`, имеющий то же значение, что и заданный экземпляр `String`
`CopyTo()`	Метод, копирующий заданное число знаков начиная с указанной позиции в этом экземпляре до указанной позиции в массиве знаков Юникода
`EndsWith()` и `StartsWith()`	Методы, определяющие, совпадает ли конец (начало) экземпляра `String` с указанной строкой
`Format()`	Метод, который заменяет каждый элемент формата в указанном объекте `String` текстовым эквивалентом значения соответствующего объекта
`IndexOf()`	Метод, возвращающий индекс первого вхождения `String` или одного или нескольких знаков в данной строке
`IndexOfAny()`	Метод, возвращающий индекс первого обнаруженного в данном экземпляре знака из

	указанного массива знаков Юникода
`Insert()`	Метод, используемый для получения копии текущей строки, содержащей добавляемые строковые данные
`Join()`	Метод, который вставляет заданный разделитель типа `String` между элементами заданного массива String, создавая одну сцепленную строку
`LastIndexOf()`	Возвращает индекс последнего вхождения указанного знака Юникода или `String` в пределах данного экземпляра
`LastIndexOfAny()`	Возвращает индекс последнего вхождения в данном экземпляре какого-либо одного или нескольких знаков, указанных в массиве знаков Юникод
`PadLeft()` и `PadRight()`	Выравнивает знаки в данном экземпляре по левому (правому) краю, добавляя справа (слева) пробелы или указанные знаки Юникода до указанной общей длины
`Remove()` `Replace()`	Методы, используемые для получения копии строки с соответствующими модификациями (при удалении или замене символов)
`Split()`	Возвращает строковый массив, содержащий подстроки данного экземпляра, разделенные элементами заданной строки или массива знаков Юникода
`Substring()`	Метод, возвращающий строку, которая представляет подстроку текущей отроки
`ToCharArray()`	Метод, возвращающий массив символов, из которых состоит текущая строка
`ToUpper()` `ToLower()`	Методы, создающие копию данной строки, представленную символами в верхнем или, соответственно, нижнем регистре
`Trim()`	Удаляет все начальные и конечные вхождения заданных наборов знаков из текущего объекта `String`
`TrimEnd()` и `TrimStart()`	Удаляет все конечные (начальные) вхождения набора знаков, заданного в виде массива, из текущего объекта String

Рассмотрим подробнее использование некоторых методов класса `String`.

Метод Split

Английское слово split означает «разбить на части». Одним вызовом этого метода можно решить популярную задачу разбиения текста на некоторые фрагменты.

Рассмотрим следующую постановку задачи:

Дан некоторый текст. Определить среднюю длину всех слов, содержащихся в этом тексте. Слова отделяются друг от друга одним или несколькими символами-разделителями, в число которых входят пробел, запятая и точка.

Очевидно, что для решения этой задачи нужно выделять в тексте отдельные слова. Это достаточно сложный циклический процесс, поскольку слова отделяются друг от друга множеством разных символов-разделителей. Именно эту задачу решает метод `Split`.

Исходную строку можно получить путём ввода с клавиатуры.

```
string text = Console.ReadLine();
```

Для успешной работы методу `Split` нужно передать параметр – массив символов-разделителей.

```
char[]  separators = new char[] {' ', ',', '.'};
```

(читатели, знакомые с массивами в языке C++ заметят, что описание и инициализация массивов в C# выглядит несколько иначе; об этом подробнее будет рассказано далее).

Метод `Split` в результате своей работы возвращает массив строк, полученных в результате выделения их из всего текста на основе информации о символах-разделителях.

```
string[] words = text.Split(separators);
```

Вот и всё! Однако, в этой короткой строчке содержится очень важное новшество, непосредственно относящее к объектно-ориентированному программированию. Метод `Split` вызывается в форме `text.Split(…)`, то есть метод `Split` как будто принадлежит строковой переменной `text`. Это действительно правильная точка зрения, которую Вы должны усвоить в ходе этого курса. Будьте внимательны, поскольку некоторые методы класса `String` вызываются «от имени» класса, например, `String.Format(…)`. Такие методы называются *статическими*. Методы, подобные `Split` называются нестатическими или *методами экземпляра* класса.

Теперь, имея массив слов текста легко окончательно решить задачу – найти среднюю длину всех слов. Для этого будет использован стандартный циклический алгоритм суммирования.

```
int L = 0;
for(int i = 0; i<words.Length; i++) L += words[i].Length;
double aL = Convert.ToDouble(L) / words.Length;
```

Здесь мы видим, что по своей форме цикл `for` в C# такой же, как и в C++. Новшеством является удобная возможность задать границу цикла (количество элементов массива!) с помощью свойства `Length` массива `words`, то есть массив сам содержит информацию о своём размере. Аналогично можно определить и длину (количество символов) одной строки: `words[i].Length`. Приведение `Convert.ToDouble` понадобилось для того, чтобы операция «деления» выдала вещественный, а не целый результат.

В C# имеется возможность несколько улучшить последний цикл, используя новую разновидность оператора цикла – `foreach`.

```
foreach(string w in words) L += w.Length;
```

Для читателя, знающего английский язык, этот цикл выглядит очень естественно, поскольку хорошо «переводится» на «человеческий» язык. В переводе на русский это звучит примерно так: «Для каждой строки `w` из `words` выполнить некоторое действие». Формальный синтаксис этого оператора следующий:

`foreach(``<тип> <переменная>` **`in`** `<массив>`**`)`** `<тело цикла>;`

Далее Вы узнаете, что кроме массивов в цикле `foreach` можно использовать и другие «контейнерные» типы.

Несмотря на правильную структуру, эта программа выдаёт неверный результат. Причина этого станет понятнее, если в цикл добавить вывод слов на экран.

```
foreach(…) { Console.WriteLine(w); L += w.Length;  }
```

Выполнение программы с входной строкой **"** `yes, no hello.` **"** покажет, что метод `Split` разбил эту строку не на три, а на шесть строк: пустая строка, “yes”, пустая строка, “no”,“hello”, пустая строка. Это произошло потому, что метод `Split` считает «словом» любую строку, расположенную между символами-разделителями. Чтобы избавиться от этой проблемы можно «заставить» метод `Split` работать несколько иначе – **не** выделять в качестве «слов» пустые строки между соседними символами-разделителями. Для этого в метод `Split` нужно передать *дополнительный* параметр - `StringSplitOptions.RemoveEmptyEntries`. Теперь вызов метода выглядит так:

```
string[] words=
 text.Split(separators, StringSplitOptions.RemoveEmptyEntries);
```

Здесь мы опять встречаемся с использованием перегруженного варианта метода, который, благодаря дополнительному параметру, исключает пустые строки.

Ещё больший интерес представляет форма записи и тип этого параметра. Тип параметра является перечислением (подробнее об этом в следующем параграфе).

Enumeration – перечислимый тип

Возможно, концепция перечислимых типов Вам уже знакома. Поэтому, ограничимся классическим, в этом случае, примером. Допустим, программа должна определять количество отработанных часов по правилу: понедельник, вторник, среда, четверг – 8 часов, пятница – 7, суббота и воскресенье – 0.

Для представления дня недели опишем перечислимый тип

```
enum WeekDay = {San, Mon, Tue, Wed, Thu, Fri, Sat};
```

Теперь функция, определяющая количество отработанных часов может выглядеть следующим образом:

```
static int WorkOurs (WeekDay wd)
{
      switch(wd)
      {
         case WeekDay.San: case WeekDay.Sat: return 0;
         case WeekDay.Mon: case WeekDay.Tue:
         case WeekDay.Wed: case WeekDay.Thu: return 8;
         case WeekDay.Fri: return 7;
      }
}
```

В последнем примере использования метода `Split` второй параметр является параметром стандартного перечислимого типа `StringSplitOptions`, определенного в .NET, а `RemoveEmptyEntries` – одним из значений этого типа.

Метод IndexOf()

Этот метод возвращает позицию первого вхождения строки, в строке, вызывающей этот метод. Если подстроки в строке нет, метод возвращает -1.

```
String s=”Hello, Helen!”;
Console.WriteLine(s.IndexOf(“He”); //выводит 0
Console.WriteLine(s.IndexOf(“he”); //выводит -1
```

Чтобы найти последующие (после первого) вхождения, можно применить перегруженную версию IndexOf, второй параметр которого указывает позицию, с которой нужно начинать поиск строки в подстроке. С помощью этого метода следующий фрагмент выводит все вхождения подстроки.

```
int k=0;
while (k!=-1)
{
  k= s.IndexOf(“He”,k);
  Console.WriteLine(k); //выводит 0
  k++;
```

```
}
```

Метод Format

Синтаксис и действие этого метода похожи на метод `Console.WriteLine`. Отличие в том, что `WriteLine` выводит сформированную строку на экран, а `Format` возвращает эту строку для дальнейшего использования в программе.

Метод `Format` является статическим и вызывается от имени класса:

```
String s=String.Format("{0}x{1}={2} и {2}={1}x{0}",2,3,2*3);
```

В результате сформирована строка “2x3=6 и 6=3*2”

Метод ToCharArray

Набор методов класса `String` не может быть идеальным средством для решения всех задач обработки текстов. Существует немало даже простых задач, которые «неудобно» решать этими методами. В этом случае остается последнее средство – решать задачу, рассматривая строку как массив символов. Однако, как уже было сказано ранее, индексированный доступ к символам строки возможен только для чтения. Именно в этом случае Вам понадобится метод `ToCharArray`, который «разбирает» целостный объект-строку на массив символов. В следующем примере решается простая задача инвертирования символов строки:

```
String s=”телефон”;
char ch;
char chAr=s.ToCharArray();
for(int i=0; i<char.Length/2; i++)
{ ch=chAr[i]; char[i]=chAr[char.Length-i-1]; chAr[char.Length-i-1]=ch; }
s=new String(char);
```

Пример демонстрирует, что метод `ToCharArray` возвращает массив символов. Последняя строка показывает, как можно создать объект-строку из массива символов.

Объектно-ориентированное программирование

Эволюция от структур к классам

Используем структуры

Напомним, как использовать традиционные уже в языке Си структурные типы и их переменные – структуры.

```
01   struct Person
```

```
02   { public string Name;
03     public double Height;
04     public double Weight;
05   }
06   class Program
07   { static void Main(string[] args)
08     { Person me, you;
09       me.Name="Это я"; me.Height=190.0;
     me.Weight=85;
10       you.Name="Это ты"; you.Height=140.0;
     you.Weight=85;
11       PersonAnalyze(me.Height,me.Weight,me.Name);
12
     PersonAnalyze(you.Height,you.Weight,you.Name);
13     }
14     static void PersonAnalyze(double h,double
     w,string n)
15     { if (h - w > 100.0) Console.WriteLine(n + "
     худой");
16       else               Console.WriteLine(n + "
     полный");
17     }
18   }
```

В определении структурного типа `Person` (стр.01-05) новым является только использование слова `public` в описании переменных. Его роль мы выясним позже.

Класс `Program` условно можно назвать главным классом, поскольку он содержит метод `Main`, с которого и начнется выполнение программы.

В методе `Main` описываются две структуры (переменные структурного типа). В отличие от языка Си, в описании структур ключевое слово `struct` не указывается. Таким образом, переменные `me` и `you` являются переменными типа `Person`.

Далее с помощью операции доступа к полю (операции точка) и операторов присваивания происходит заполнение переменных `me` и `you` информационным содержимым (стр.09-10).

Наконец, в стр. 11 и 12 вызывается метод `PersonAnalyze` класса `Program` сначала с данными структуры `me`, а затем с данными структуры you. Заметим, что метод `PersonAnalyze`, как и метод `Main` описан как `static`. На экране должен появиться следующий результат:

```
Это я полный
Это ты худой
```

Структурный тип как параметр

Дальнейшее использование структурного типа повышает уровень Вашего программного кода. Например, применение параметров структурного типа делает функции (методы) более естественными:

```
static void PersonAnalyze(Person p)
{ if (p.Height-p.Weight> 100.0)
    Console.WriteLine(p.Name + " худой");
  else
    Console.WriteLine(p.Name + " полный");
}
```

Использование такого метода в Main подчеркивает, что Ваша программа оперирует сущностями предметной области, а не примитивными порциями данных:

```
PersonAnalyze(me);
PersonAnalyze(you);
```

Помещаем метод в структурный тип

В предыдущем примере метод `PersonAnalyze` имеет довольно слабое отношение к структурному типу `Person` (только название). Основной причиной этого является размещение метода за пределами структурного типа `Person`.

Проделаем следующую трансформацию программы:

```
01   struct Person
02   { public string Name;
03     public double Height;
04     public double Weight;
05     public void PersonAnalyze()
06     {
          if (Height-Weight>100.0)
               Console.WriteLine(Name+" полный");
07        else Console.WriteLine(Name + " худой");
08     }
09   }
10   class Program
11   { static void Main(string[] args)
12     { Person me, you;
13       me.Name = ""Это я"; me.Height = 190.0;
     me.Weight=85;
14       you.Name="Это ты"; you.Height=140.0;
     you.Weight=85;
15       me.PersonAnalyze();
16       you.PersonAnalyze();
```

```
17      }
18    }
```

Первое, что следует отметить – метод `PersonAnalyze` стал частью структурного типа `Person`. При этом в его описании исчезло слово `static`. Это означает, что вызов метода будет осуществляться структурной переменной этого типа. В стр. 15 и 16 мы видим два таких вызова. Важно, что в методе `PersonAnalyze` пропали параметры. Когда переменная `me` вызывает метод `PersonAnalyze`, нет необходимости также передавать дополнительные данные через параметры – необходимые величины находятся в полях структурного типа и доступны методу. Кроме того, вызовы `me.PersonAnalyze` и `you.PersonAnalyze` дадут различные результаты, поскольку используют различные данные двух различных структур.

Теперь тип `Person` стал более полноценным, поскольку определяет не только данные, имеющие отношение к человеку, но и некоторые его возможности в виде методов.

Превращение в класс

Наконец, рассмотрим последнюю модификацию программы.

```
01   class Person
02   { public string Name;
03     public double Height;
04     public double Weight;
05     public void PersonAnalyze()
06     { if (Height-Weight>100.0)
     Console.WriteLine(Name+" худой ");
07       else Console.WriteLine(Name + " полный ");
08     }
09   }
10   class Program
11   { static void Main(string[] args)
12     { Person me;
13       me = new Person();
14       Person you = new Person();
15       me.Name="Это я"; me.Height=190.0;
     me.Weight=85;
16       you.Name="Это ты"; you.Height=140.0;
     you.Weight=85;
17       me.PersonAnalyze();
18       you.PersonAnalyze();
19     }
20   }
```

Во-первых, в заголовке структурного типа слово `struct` заменено ключевым словом `class`. Как следствие, в методе `Main` уже недостаточно только описать переменые. Переменные, порождаемые на основании класса, называются объектами и требуют обязательного создания с помощью операции `new`. Мы видим, что эту операцию можно выполнить в отдельном операторе присваивания (стр.13) и в момент описания переменной с инициализацией (стр.14).

Пока разница между структурными типами и классами не очень заметна. После прочтения этого пособия Вы сможете аргументировано оценить эту разницу самостоятельно.

Классы и объекты

Теперь мы можем дать предварительное определение понятия класс. Класс – это программная конструкция, определяющая новый тип данных. Для этого в классе определяются переменные и методы. Класс является «правилом» для создания объектов (экземпляров этого класса). Все объекты имеют одинаковый набор переменных. Однако соответствующие переменные различных объектов независимы друг от друга.

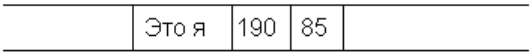
Далее можно обращаться к переменным объекта и вызывать методы объекта. Обратите внимание на словосочетание «переменные объекта» - действительно, каждый объект класса имеет свой собственный комплект переменных, описанных в его классе. Множество значений переменных объекта можно называть состоянием объекта. Иначе обстоит дело с методами – они существуют в одном экземпляре и все объекты класса пользуются методами «сообща». Множество методов определяет поведение объектов класса.

Значимые и ссылочные переменные

В C# все переменные можно разделить на две категории – переменные значимого и ссылочного типа.

Значимая переменная хранит свое значение непосредственно в выделенной ей компилятором памяти. Структуры являются значимыми переменными, поэтому размещение в памяти переменной `me` в вариантах программ, где она была структурой, выглядит так:

Память, выделенная
переменной me



Значимыми являются также переменные основных встроенных типов данных – числовые (`double`, `int`), символьные (`char`), логические (`bool`). А вот переменные строкового типа (`String`) – ссылочные.

Ссылочная переменная в своей памяти хранит адрес (ссылку) на другое место в памяти, где хранятся данные. Помимо решения проблем с передачей изменяемых параметров, эта двухуровневая схема адресации в большей степени соответствует духу объектно-ориентированного подхода. Вспомним вариант программы, где `me` была объектом класса `Person`. Переменные, предназначенные для указания на объекты классов, являются ссылочными переменными, поэтому схема размещения в памяти такова:

Теперь размер памяти, выделяемой любой ссылочной переменной, одинаков – это размер, достаточный для хранения адреса. Данные, сгруппированные в виде информационного объекта, находятся в том месте, на которое указывает адрес. Теперь становится понятнее, зачем объекты необходимо создавать с помощью операции `new`. Компилятор не занимается выделением памяти для объектов. Эта операция должна быть выполнена динамически, то есть во время выполнения программы. Если Вы забудете осуществить выделение памяти операцией `new` и начнете использовать такую переменную, то в программе произойдет ошибка времени выполнения «null reference».

Уточним, какие переменные в C# являются значимыми, а какие – ссылочными.

Значимые переменные	Ссылочные переменные
Переменные встроенных типов Структуры, не использующие для создания операцию `new`	Массивы Структуры, создаваемые с помощью new Объекты класса String Объекты классов

Принадлежность переменной к категории ссылочной или значимой влечет целый ряд последствий, рассматриваемых далее подробнее.

Если переменная – локальная (описана внутри метода класса), то после вызова метода, память, выделенная такой переменной, автоматически освобождается. Однако, если эта локальная переменная является ссылочной, то важно понять, что происходит с памятью, выделенную под адресуемый ею

объект. Автоматически освобождать ее при выходе из метода еще нельзя – возможно на этот объект ссылается другая переменная программы.

Разместим в классе `Program` рядом с методом `Main` еще один метод `Grow`, увеличивающий рост человека, переданного в качестве параметра:

```
public static void Grow(Person p) //этот метод мог быть проще
{Person local; local=p; local.Weight++;}
```

Причину появления в заголовке метода ключевого слова `static` Вы узнаете позже.

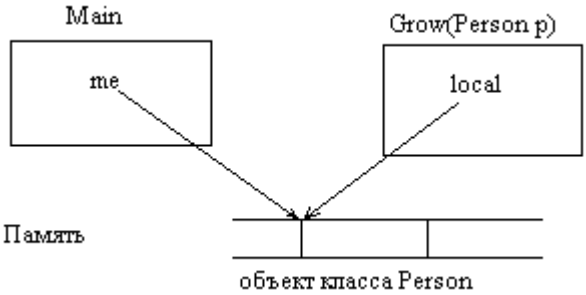
Перед вызовом этого метода в `Main` должен быть создан объект `Person`.

```
Person me = new Person();
me.Name = "Это я"; me.Height = 190.0; me.Weight = 85;
```

Далее осуществляется вызов метода Grow:

```
Grow(me);
```

В процессе выполнения метода `Grow` создается локальная переменная `local`, которая, благодаря присваиванию `local=p;` также ссылается на объект `Person`.



После выполнения метода `Grow` переменная `local` исчезает, однако объект `Person` в памяти остается и к нему имеется возможность доступа через переменную `me`. Таким образом, благодаря ссылочным переменным легко решается проблема изменяемых параметров.

Теперь зададим себе вопрос – что если переменная `me` также исчезает, освобождая занимаемую ею память? В этом случае становится невозможным доступ к объекту `Person`, занимающему свой участок памяти. В этой ситуации возникает опасность “утечки памяти” - в процессе выполнения программы может возникнуть много неиспользуемых объектов. В некоторых языках решение этой проблемы возлагалось на программиста. Он должен был предусмотреть явное уничтожение объекта специальным методом-деструктором.

Однако в современных языках, в частности и в C#, используется другой подход. Во время выполнения программы в фоновом режиме выполняется специальная утилита – сборщик мусора (garbage collector), который автоматически уничтожает объекты, для которых не осталось ссылок в программе.

Отметим еще несколько особенностей.

При выполнении присваивания для значимых переменных-структур происходит поэлементное копирование, а для ссылочных переменных на объекты – только копирование адреса.

Операции сравнения для ссылочных переменных обычно реализованы как сравнение адресов объектов, на которые они ссылаются. Поэтому имеют смысл обычно только операции == (равно) и != (не равно). Однако есть возможность самостоятельно переопределить операции сравнения для реализации более содержательного сравнения, основанного на состоянии объекта.

Механизм скрытой передачи адреса на объект решает большую часть проблем, возникающих при передаче параметров. В языке C и его «наследниках» передача параметров (то есть передача значений фактических параметров в формальные переменные) осуществляется только по значению. В этом случае функция, которая «возвращает» результат своей работы через один или несколько параметров, должны были как-то обходить это правило. Понятно, что значимые переменные уже обладают такой способностью. Однако некоторые ситуации таким способом не учитываются.

Сначала рассмотрим две следующие ситуации:

1) Необходимо обеспечит передачу «по ссылке» значимой переменной.
2) Необходимо обеспечить изменение методом самой ссылки (адреса).

В обеих ситуациях можно воспользоваться специальным видом параметров – `ref`-параметрами. Для этого нужно указать ключевое слово `ref` в заголовке метода перед определением формального параметра и при вызове метода перед именем фактического параметра:

```
public void DoSomething(ref Person p, ref int i)
{ Person newMe= new Person();
  newMe.Name="Это я"; newMe.Height=190.0;
newMe.Weight=85;
  me=newMe;
. i++;
}
. . .
int k=5;
DoSomething(ref me, ref k);
```

Здесь в методе DoSomething обеспечивается передача по ссылке как ссылочной переменной me типа Person, так и значимой переменной k типа int. Благодаря этому, после вызова метода переменная me ссылается на новый объект, а переменная k изменяет свое значение.

Еще одна ситуация, представляющая интерес - передача в метод неинициализированные переменные.

Инициализация переменных перед их использованием является обязательным требованием C#. Таким образом, компилятор следит за тем, чтобы ссылочные переменные в момент их использования указывали на некоторый объект. В противном случае они считаются неинициализированными и имеют специальное знначнение null. Однако в некоторых случаях это требование становится неудобным. Что, если первоначальное значение для переменной может быть определено только в результате выполнения достаточно сложного метода? В этом случае нужно использовать специальные out-параметры. Ключевое слово out следует указывать, как и слово ref перед формальными и фактическими параметрами.

```
class Program
{ static void Main(string[] args)
  { Person me;
    MakePerson(out me);
    me.PersonAnalyze();
  }
  public static void MakePerson(out Person p)
  { p = new Person();
    p.Name="Это я"; p.Height=190.0; p.Weight=85;
  }
}
```

Здесь мы видим, что переменная me, описанная в методе Main, используется в качестве параметра метода MakePerson. На момент вызова эта переменная не ссылается на некоторый созданный объект. Для ссылочных переменных это и означает, что переменная не инициализирована. Однако создание объекта и связывание его с переменной успешно происходит методе MakePerson с out-параметром.

Для значимых переменных использование out-параметров не столь важно, поскольку значимые переменные инициализируются автоматически нулевыми значениями.

Конструкторы класса

Рассмотрим подробнее, как создается объект класса Person:

```
newMe = new Person();
```

Сначала в правой части присваивания выполняется операция `new`, которая резервирует в памяти участок, способный хранить все переменные класса. Однако назвать это действие полноценным созданием объекта нельзя. Здесь не хватает того, что происходит и в реальной жизни – при рождении объект не только занимает место в пространстве, но и получает полный набор значений своих характеристик (начальное состояние объекта). Это необходимо выполнить и в момент создания объекту. Вот почему после операции `new` указывается не просто тип Person, а вызывается специальный метод, имя которого совпадает с именем класса. Такой метод называется конструктором. Исходя из такой роли конструктора, он должен быть определен в каждом классе.

Конструктор класса имеет несколько синтаксических особенностей:

1. Обычно (но не всегда!) конструктор описывается как `public`.
2. При определении конструктора в заголовке не указывается тип возвращаемого значения. Конструктор в принципе ничего не может возвращать, поэтому даже ключевое слов `void` здесь будет неуместно.
3. Имя конструктора всегда совпадает с именем класса.

Обсудим теперь список параметров конструктора. Логично, что через фактические параметры при вызове конструктора должны быть указаны данные, позволяющие определить состояние объекта. Поэтому, например, для создания полноценного объекта класса `Person` можно указать параметры для имени, веса и роста:

```
public Person(string n, double h, double w)
{ name=n; Height=h; Weight=w; }
```

При желании мы можем использовать имена параметров, совпадающие с именами переменных классов. Однако в этом случае придется использовать ключевое слово `this` для решения проблемы коллизии имен (совпадения имен двух переменных в одной области видимости):

```
public Person(string Name, double Height, double
   Weight)
{ this.Name= Name; this.Height= Height; this.Weight=
   Weight; }
```

Слово `this` указателем на текущий объект. Это слово обозначает объект данного класса, который вызвал метод.

Список параметров не обязательно должен соответствовать набору переменных класса. Допустим, рост и вес определяются на основании возраста по таблице стандартных соотношений роста и веса:

```
public Person(int age)
{ Height=table[age].height;Weight= table[age].weight; }
```

Заметим, что этот конструктор не обеспечивает назначение объекту имени

```
Person p = new Person(10);
p.PersonAnalyze();
```

Такой эксперимент покажет, что объект, на который ссылается переменная p, имеет имя “”, то есть пустую строку. Это стандартное «нулевое» значение, которое автоматически присваивается строковым переменным, если это инициализация не была выполнена явно. Для переменных числовых типов таким стандартным значением является 0, а для логических переменных - false.

В классе может быть одновременно несколько конструкторов, которые должны отличаться друг от друга сигнатурой (последовательностью типов параметров). Такое явление называется перегрузкой. В зависимости от контекста может требоваться различная инициализация переменных объекта. Перегрузка конструкторов и обеспечивает решение этой задачи. Перегрузка допустима и для других методов класса.

В классе можно объявить статический конструктор с атрибутом static. Он вызывается автоматически - его не нужно вызывать стандартным образом. Точный момент вызова не определен, но гарантируется, что вызов произойдет до создания первого объекта класса. Такой конструктор может выполнять некоторую предварительную работу, которую нужно выполнить один раз, например, связаться с базой данных, заполнить значения статических полей класса, создать константы класса, выполнить другие подобные действия. Статический конструктор, вызываемый автоматически, не должен иметь модификаторов доступа. Вот пример объявления такого конструктора в классе Person:

```
static Person()
{ Console.WriteLine("Выполняется статический
конструктор!"); }
```

Особую роль играет конструктор без параметров. Его называют конструктором по умолчанию, поскольку он может использовать для инициализации переменных объекта некоторые стандартные значения. Мы можем определить, например, такой конструктор:

```
public Person( ){ name=”noname”; Height=50; Weight=4; }
```

Если в классе явно не определен ни один конструктор, то конструктор по умолчанию генерируется компилятором. Однако ничего интересного такой конструктор не выполняет – он инициализирует переменные объекта стандартными «нулевыми» значениями. Если в составе класса имеется переменная, являющаяся объектом некоторого класса, то в этом случае она будет иметь значение null – специальное слово, обозначающее отсутствие ссылки на объект в памяти.

Заметьте, что если программист сам создает один или несколько конструкторов, то автоматического добавления конструктора по умолчанию не происходит.

Статические элементы

В ходе изучения данного пособия Вы должны усвоить объектно-ориентированный стиль программирования. Кратко это можно описать таким образом.

1. Сначала создайте классы – полезные правила для создания объектов.
2. Далее создавайте объекты – экземпляры этих классов и заставляйте их выполнять нужную Вам работу.

Таким образом в процессе выполнения программы непосредственно действуют объекты. Однако в некоторых случаях естественнее считать, что действия выполняются самим классом. Хорошим примером является класс Math – все его методы вызываются от имени класса:

```
Console.WriteLine(Math.Sin(0.5)+Math.Cos(0.5));
```

Было бы довольно странно для вычисления математических функций сначала создавать объект-«математику»:

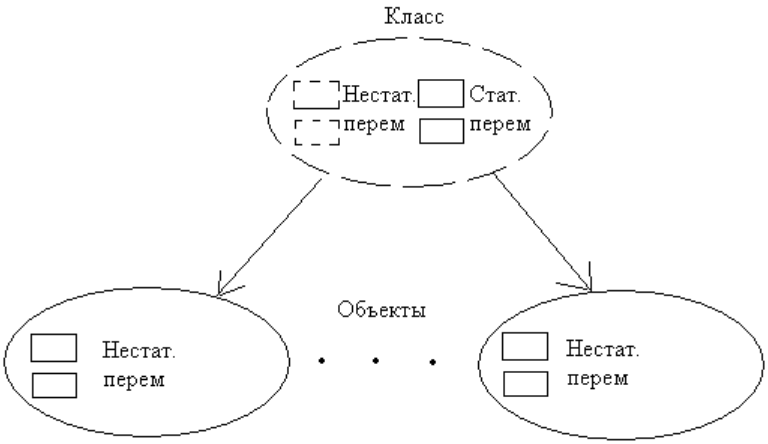
```
Math m1=new Math();
Console.WriteLine(m1.Sin(0.5)+m2.Cos(0.5));
```

Это означало бы, что можно создавать несколько «математик». Но ведь они все должны действовать абсолютно одинаково.

Аналогично дело обстоит и с переменными – бывают случаи, когда некоторые данные естественнее представлять не как порцию информации, принадлежащую объекту, а классу в целом. В классе Math эта ситуация встречается при доступе к тригонометрической константе π - `Math.PI`.

Элементы класса (переменные и методы), которые соотносятся не с объектами, а с классом, называются статическими. В описании статических элементов используется ключевое слово `static`. В отличие от статических членов обычные элементы класса называются элементами экземпляров класса – методы экземпляров и переменные экземпляров.

Переменные экземпляра описываются в классе и хранятся в объектах класса. Статические переменные описываются и хранятся (в единственном числе) в классе, как показано на следующем рисунке:



Рассмотрим следующий пример. Класс `Person` моделирует ситуацию, когда множество людей располагают некоторым общим запасом пищи и делят его поровну. Количество пищи `FoodQuantity` и количество людей `PeopleCount` – характеристики всей совокупности людей. Поэтому они описаны как статические переменные. Метод `Description` описывает состояние всего множества людей и также является статическим методом.

```
class Person
{ public static double FoodQuantity = 100.0;
  public static int PeopleCount = 0;
  public double Weight;
  public Person()
  { Weight = 10.0; PeopleCount++; }
  public void ToEat()
  { double t = FoodQuantity / 2.0 / PeopleCount;
    Weight += t; FoodQuantity -= t;
  }
  public static string Description()
 { return String.Format("Людей - {0} Еды - {1}",
    PeopleCount,FoodQuantity);
  }
}
class Program
{ static void Main(string[] args)
  { Person p1 = new Person(); p1.ToEat();
    Console.WriteLine("Вес p1 - {0}", p1.Weight);
```

```
    Console.WriteLine(Person.Description());
    Person p2 = new Person(); Person p3 = new Person();
    p1.ToEat(); Console.WriteLine("Вес p1 -
{0}",p1.Weight);
    Console.WriteLine(Person.Description());
  }
}
```

Статические методы могут использовать только статические переменные и другие статические методы класса. Попытка использовать в методе `Description` переменную `Weight` привела бы к ошибке компилятора.

Для инициализации статических переменных можно использовать статический конструктор:

```
static Person()
{ FoodQuantity = 100.0; PeopleCount = 0; }
```

Если в начале изучения языка C# не сразу используются его объектно-ориентированные возможности, структура программы выглядит примерно следующим образом:

```
class Program
{ static void Main(string[] args)
  { Do1();
    Console.WriteLine(Do2());
  }
  static void Do1() { Console.WriteLine("метод Do1"); }
  static string Do2() { return "метод Do2"; }
}
```

Таким образом, можно не обращать внимания на наличие в программе класса `Program`. Программа сстоит из нескольких статических методов, которые вызываются в «главном» методе `Main`.

Генерация случайных чисел

Необходимость в создании последовательности случайных чисел возникает в программировании довольно часто. Кроме того, способ, которым это делается в C#, характерен с точки зрения объектно-ориентированного подхода. В других языках программирования для генерации случайных чисел имеется некоторая функция (например, `Random` в языке Pascal). Следует учитывать детерминированную природу алгоритма генерации случайных чисел. Обычно такие алгоритмы основаны на некотором перемешивании цифр начального числа («зерна»). Для того чтобы серия случайных чисел каждый раз была другой, в качестве начального зерна алгоритму следует передавать различные значения начального зерна. Для этого в языке Pascal

имеется функция Randomize, использующая в качестве начального зерна текущее системное время.

Однако в C# нет самостоятельных функций, а только методы классов. В качестве ближайшей аналогии функциям Random и Randomize можно использовать статические методы некоторого класса. Однако на самом деле в C# имеется класс Random с набором методов для генерации случайных чисел. Таким образом, фрагмент программы, генерирующий два случайных числа, может выглядеть так:

```
Random rnd=new Random();
//целое случайное число в диапазоне от 1 до 6
int i=rnd.Next(1,7);
//целое вещественное число в диапазоне от 0 до1
double d=rnd.NextDouble();
```

Таким способом решается и проблема уникального начального зерна – при создании нового объекта в качестве начального зерна неявно используется текущее системное время. Тем не менее, класс Random следует использовать внимательно. Следующий фрагмент продемонстрирует две одинаковые серии случайных чисел – i11 будет равно i21, а i12 – равно i22.

```
Random rnd1=new Random();
Random rnd2=new Random();
int i11=rnd.Next(1,7);
int i12=rnd.Next(1,7);
int i21=rnd.Next(1,7);
int i22=rnd.Next(1,7);
```

Дело в том, что недостаточная точность измерения системного времени при создании объектов rnd1 и rnd2 привела к созданию идентичных объектов. Для создания различных объектов необходимо обеспечить достаточный интервал между их созданием.

Массивы в языке C#

Хотя основные приемы использования массивов C# унаследовал от C++, следует обратить внимание на ряд важных особенностей.

Каждый массив является объектом класса System.Array. Поэтому в жизненном цикле массива имеется стадия описания массива и стадия создания массива. Внимание – в описании массива не указывается размер (количество элементов):

```
int [] Arr1;
Person [] Arr2;
```

Как и раньше, массив – это коллекция однотипных элементов. Массив Arr1 будет содержать целые числа, а массив Arr2 – объекты класса

Person. Еще отметим «перемещение» пары квадратных скобок – они в C# указываются перед именем массива. Таким образом, конструкция «int []» является полноценным описателем типа массива.

Далее массив можно создать и на этой стадии нам понадобится операция new:

```
Arr1 = new int[10];
```

Использование new почти не изменилось – после new нужно указать тип объекта, а у нас это int[]. Только теперь в квадратных скобках нужно указать количество элементов массива. Кроме того, отсутствуют скобки со списком параметров.

Дальнейшее использование массива может происходить обычным образомю Например:

```
for (int i=0; i<10; i++) Arr1[i] = i*2;
```

Как видите, нумерация, по прежнему начинается с 0.

Аналогично поступим с массивом Arr2:

```
Arr2 = new Person[Arr1[3]];
```

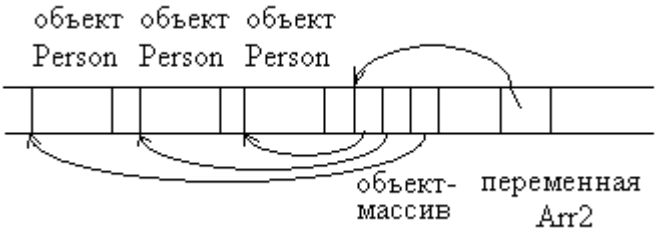
Здесь проявилась замечательная особенность массивов в C# - их размер может задаваться выражением, значение которого определится только во время выполнения программы. Более того, Вы можете заново создать массив:

```
Arr2 = new string[Arr2.Length + 2];
```

Здесь мы воспользовались свойством Length класса System.Array. В результате размер нового массива больше на 2 элемента размера старого. Но учтите, что «новый» массив не содержит элементов старого массива – ведь это совсем новый объект в новом месте памяти.

Отметим, что в примере мы не инициализировали массив. В отношении массива требование инициализации не действует. Дело в том, что при создании массива с помощью new создается множество ссылок, каждая из которых содержит «пустой» указатель null. Этого достаточно, чтобы C# позволил приступить к использованию массива. Однако здесь появляется возможность для ошибки во время выполнения массива – если Вы попытаетесь использовать объект, на который ссылается такая null-ссылка. Поэтому нужно выполнить что-то в таком духе:

```
for (int i=0; i<Arr2.Length; i++) Arr2[i]=new Person();
```



Поскольку массив – это объект специального типа, его можно использовать как параметр метода или как тип возвращаемого значения. Например, следующий метод `ModifyArray` принимает любой целочисленный массив в качестве параметра и возвращает целочисленный массив вдвое большего размера, первая половина которого заполнена данными массива-параметра, а вторая – нулями.

```
static int[] ModifyArray(int[] inArr)
{ int [] rArr = new int[inArr.Length * 2];
  for (int i = 0; i < inArr.Length; i++)
  { rArr[i] = inArr[i]; rArr[inArr.Length + i] = 0; }
  return rArr;
}
```

Этот метод можно использовать следующим образом:

```
int[] Arr3 = ModifyArray(Arr1);
```

Заметим, что таким образом мы можем имитировать полную динамичность массива, как множества элементов – добавлять в него новые элементы уже после создания массива или удалять существующие. Однако более эффективня реализация таких гибких структур данных достигается с помощью других контейнерных классов, оо которых Вы узнаете дальше.

Следует отметить, что кроме обычных приемов работы с массивами (обращение к элементам с помощью индекса, циклы и т.д.) класс `System.Array` предоставляет ряд дополнительных и весьма полезных методов. Некоторые из них перечислены в следующей таблице:

Метод	Описание
`static int IndexOf (Array array, Object value)`	Возвращает первое вхождение значения `value` в массив `array`. Если `array` не содержит заданного значения, метод возвращает отрицательное целое число.
`public static void Sort (Array array)`	Сортирует элементы во всем одномерном массиве `array`.
`static int BinarySearch`	Быстрый поиск методом

`(Array array, Object value)`	половинного деления позиции значения `value` в объекте `array`. Перед вызовом этого метода объект `array` необходимо отсортировать. Если `array` не содержит заданного значения, метод возвращает отрицательное целое число.

Многомерные массивы

Рассмотрим только 2-х мерные массивы. Использование массивов большей размерности принципиально не отличается. Существуют два вида таких массивов – прямоугольные и «рваные» (ступенчатые).

У прямоугольного массива все строки имеют одинаковое количество элементов (также как и столбцы). Такая структура в математике называется двумерной матрицей.

Описание двумерного массива выглядит следующим образом:

```
int [ , ] matrix;
```

При создании такого массива, как обычно, используется операция `new`:

```
matrix = new int[3,4];
```

Здесь создается прямоугольный массив из 3-х строк и 4-х столбцов. Дальнейшее использование такого массива вполне традиционно. Например, можно заполнить весь массив содержимым следующим образом:

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

Обратим внимание, что для определения количества шагов в циклах перебора вместо свойства `Length` (общее количество элементов), нужно использовать метод `GetLength` с параметром – номером измерения.

Другой тип многомерных массивов – рваные массивы – следует представлять как одномерный массив, элементами которого являются в свою очередь, массивы. Описание такого массива несколько отличается:

```
int [][] jagArray;
```

Здесь используются две пары квадратных скобок. Создание «рваного» 2-х мерного массива состоит из двух этапов. Сначала создается главный массив:

```
jagArray = new int[5][];
```

Это можно понимать как создание 5-элементного массива, элементами которого будут являться пока не созданные объекты-массивы (пустая пара квадратных скобок). На следующем этапе нужно содать и эти массивы, например, следующим образом:

```
for (int i=0; i<jagArray.Length; i++)
  jagArray[i]=new int[i+7];
```

При переборе ступенчатого массива следует учитывать, что не все элементы главного масива существуют:

```
int s=0;
for (i=0;i< jagArray.Length; i++)
 if jagArray [i]!=null
   for (j=0; j< jagArray [i].Length; j++)
     s=s+ jagArray [i][j];
```

Класс ArrayList

Несмотря на новые возможности массивов в C#, их еще нельзя назвать полностью динамическими. После создания массива операцией `new` количество элементов в массиве будет зафиксировано и не может быть изменено в ходе выполнения программы. Если же Вы попытаетесь создать массив еще раз с другим количеством элементов, то это будет новый массив, не содержащий старых значений, которые хранятся в старом массиве.

Если в Ваших задачах требуется динамическое изменение размера масива, можно использовать стандартный класс `ArrayList` из пространства имен `System.Collections`.

У класса `ArrayList` есть еще одно важное отличие от массивов – он способен хранить элементы совершенно произвольного типа.

Для использования `ArrayList` в программе нужно подключить пространство имен `System.Collections`.

Описание и создание объекта `ArrayList` происходит как обычно:

```
ArrayList persons=new ArrayList();
```

При этом используется конструктор по умолчанию, который создает объект `ArrayList`, не содержащий ни одного элемента.

Теперь с помощью метода `Add` мы можем добавлять в persons элементы:

```
Person p=new Person();
persons.Add(p);
persons.Add(new Person());
```

Здесь мы добавили в `persons` два объекта `Person` – один объект, на который ссылается переменная p, а второй объект – безымянный (для него не существует переменной). К первому объекту можно получать доступ через «его» переменную p и через объект `persons`, второй объект доступен только через `persons`.

Возникает вопрос – в каком порядке находятся элементы внутри `persons`. Ответ интуитивно ясен – в порядке их добавления. Теперь

использовать содержимое persons можно так же как и для обычного массива, например:

```
for (int i=0;i<persons.Count;i++)
persons[i].PersonAnalyze();
```

Обратите внимание на то, что для определения количества элементов в ArrayList используется не свойство Length как у массивов, а свойство Count.

Выполнение такого цикла приведет к ошибке компиляции:

```
'object' does not contain a definition for
'PersonAnalyze'
```

Компилятор «говорит», что в классе Object не определен метод PersonAnalyze. Откуда взялся класс Object? Дело в том, что ArrayList является универсальным контейнером, способным хранить объекты любого типа. Платой за это является потеря информации о действительном типе объекта, когда мы обращаемся к нему как к элементу ArrayList. Все, что известно о типе этого объекта – он является любым объектом, то есть объектом класса Object – общем предке всех классов .NET. И именно об отсутствии метода PersonAnalyze в классе Object сообщает компилятор.

На практике мы обычно знаем, какого типа объект находится в ArrayList. В этих случае вполне оправдан риск явного приведения к этому типу:

```
for (int i=0;i<persons.Count;i++)
  ((Person)persons[i]).PersonAnalyze();
```

Намного сложнее дело обстоит, если Вы хотите хранить в ArrayList разнотипные объекты и заранее не известен порядок их следования. Как решать такие задачи Вы узнаете позже.

Метод Remove позволяет удалить объект из ArrayList:

```
persons.Remove(p);
```

Если параметр-объект не содержится в ArrayList, то метод Remove не имеет никакого эффекта.

Отметим еще несколько полезных методов:

RemoveAt	удаление объекта с указанной позицией
Insert	вставка объекта в указанную позицию
Sort	упорядочивает элементы в ArrayList
Clear	удаление всех элементов из ArrayList
Contains	определяет, содержится ли объект в ArrayList
IndexOf	возвращает позицию объекта в ArrayList

Хотя элементы в ArrayList находятся в порядке возрастания их номеров (и обычно в порядке их добавления), во многих случаях этот порядок не имеет значения. Допустим, нужно определить суммарный вес объектов Person в ArrayList. Для этого нужно просмотреть все объекты в ArrayList в любом порядке. Для таких ситуаций очень удобна новая разновидность оператора цикла:

```
double s = 0;
foreach (Person pers in persons) s = s + pers.Weight;
```

В заголовке цикла описывается переменная, которая будет использоваться для перебора (Person p) и указывается место, где осуществляется перебор (in persons). Всю остальную работу по организации перебора цикл foreach выполняет автоматически. Заметьте, что явное приведение к типу Person здесь выполнено путем описания переменной цикла p как Person.

Цикл foreach можно использовать и для обычных массивов.

Несмотря на такую простоту, использование цикла foreach ограничено следующим фактом – в цикле foreach доступ к элементам массива или ArrayList может происходить только для чтения.

Класс List<>

Класс List<> является аналогом ArrayList, но позволяет хранить только объекты заданного типа. Тип хранимых объектов указывается при описании в угловых скобках <>:

```
List<int> integers = new List<int>(); //множество целых чисел
List<Person> persons; //множество людей
persons= new List<Person>();
```

Теперь у Вас нет возможности нарушить строгую типизацию:

```
integers.Add(new Person());  //ошибка компиляции
```

Для использования класса List<> нужно подключить пространство имен System.Collections.Generic.

Инкапсуляция

До сих пор переменные и методы наших классов описывались с модификатором доступа public. Это позволяло использовать их за пределами класса. Например, для увеличения роста человека, на которого ссылается переменная p можно написать оператор:

```
p.Height++;
```

Однако в современном объектно-ориентированном программировании действует правило инкапсуляции, согласно которому все переменные класса

делаются закрытыми (private), то есть недоступными за пределами класса. Доступ к этим переменным осуществляется через открытый интерфейс – открытые методы класса.

В связи с этим внесем изменения в класс Person:

```
class Person
{ private string name;
  private double height;
  private double weight;
  public Person(string Name, double Height, double
Weight)
  { name=Name; height=Height; weight=Weight; }
  public Person( ){ name="noname"; height=50; weight=4;
}
  public void PersonAnalyze()
  { if (height-weight>100.0)
      Console.WriteLine(name+" полный");
    else
      Console.WriteLine(name + " худой");
  }
}
```

Отметим два вида изменений:

- все переменные класса описаны с использованием модификатора доступа private;
- имена этих переменных начинаются с маленькой буквы. Хотя это и не является строгим правилом языка, однако большинство программистов делают так, чтобы уже по имени переменной определить ее закрытый статус.

Теперь за пределами класса уже нельзя осуществить непосредственное использование переменных. Однако выход есть (и даже несколько). Например, можно создать в классе два открытых метода для доступа к каждой переменной (их еще называют методами-аксессорами или get- и set- методами):

```
public double GetHeight() {return height; }
public void SetHeight(double newHeight) {
height=newHeight; }
```

Теперь для увеличения роста человека на единицу потребуется два вызова методов:

```
p.SetHeight(p.GetHeight()+1);
```

Главный вопрос здесь – зачем нужно такое ограничение? Ведь последняя строка не только «ужасно» выглядит, но и замедляет выполнение

программы. Дело в том, что преимущества инкапсуляции намного важнее, чем упомянутые здесь недостатки.

Вспомним, что важнейшей целью объектно-ориентированного программирования является уменьшение разрыва в понятиях программной реализации и предметной области, чтобы сделать процесс программирования похожим на моделирование, использующее элементы предметной области.

Если оператор

```
p.Height++;
```

еще может соответствовать реальному процессу (увеличение роста человека), то как можно содержательно трактовать оператор

```
p.Height--;
```

или

```
p.Height=-10;
```

Таким образом, непосредственное использование переменных не способствует поддержке правил и ограничений предметной области (так называемых бизнес-правил). Эту проблему можно решить несколькими способами.

1. Использование специализированных методов, соответствующих содержательным действиям в предметной области. Например, метод `Grow` в классе `Person` может реализовать естественный рост человека на протяжении некоторого периода:

```
public void Grow(int days) { . . . }
```

Реализация такого метода может быть достаточно реалистичной, учитывая возраст человека.

2. Реализацией ограничений в методах доступа:

```
public void SetHeight (double newHeight)
{ if ((newHeight>0)&&( newHeight<230)&&
(newHeight>height))
   height=newHeight;
}
```

3. Реализацией методов-свойств. О свойствах подробнее будет рассказано ниже.

Обратим внимание на важные следствия такого подхода. Класс с его методами и переменными становится в достаточной степени черным ящиком. Пользователю класса не известны ни особенности реализации методов класса, ни даже информационная структура класса. Это позволяет разделить программный проект на разные по роли фрагменты, которые часто взаимодействуют по принципу клиент-сервер. Клиент использует класс, зная его открытый интерфейс. Примером клиента является метод `Main`, использующий встроенные и пользовательские классы для решения конкретной задачи. Сервер – это класс, предоставляющий свои услуги.

Разработчик серверного класса может изменять (совершенствовать) детали его устройства и функционирования, пока это не влияет на открытый интерфейс класса.

Обработка ошибок

Вернемся к рассмотрению метода класса `Person`, который обеспечивал некоторые ограничения рассматриваемой предметной области.

```
public void SetHeight (double newHeight)
{ if ((newHeight>0)&&( newHeight<230)&&
(newHeight>height))
   height=newHeight;
}
```

Использование этого метода с некорректными данными никак не влияет на состояние объекта `Person` (у `if`-оператора нет `else`-части). Как ни странно, это приводит к еще худшим последствиям. Предположим, что программа выполнила оператор `p.SetHeight(-10)`. После выполнения метода объект остается в прежнем состоянии, и программа продолжает «корректно» работать. Теперь трудно будет обнаружить, почему дальнейшее использование такого объекта приводит к ошибочным последствиям.

При отсутствии контроля дальнейшее использование объекта человек, скорее всего привело бы к аварийному завершению программы (например, попытка вычислить квадратный корень высоты в более точных формулах определения полноты человека).

Попробуем улучшить реализацию метода `SetHeight`, разместив в части `else` оператор вывода диагностического сообщения:

```
public void SetHeight (newHeight)
{ if (newHeight>0)&&( newHeight<230)&&
(newHeight>height)
   height=newHeight;
  else Console.WriteLine("Ошибка: недопустимый рост");
}
```

Это не улучшает ситуацию. Во-первых, при использовании в рамках консольного приложения, пользователь программы может просто не заметить дополнительной строчки, выводимой на экран – программа «успешно» продолжает работать и выводить другую информацию. Во-вторых, такой класс `Person` нельзя использовать в оконных приложениях, где нельзя использовать класс `Console`.

Все это означает, что в `else`-части нужно осуществлять завершение программы. Однако в C# нет оператора, позволяющего это сделать в любом месте программного кода. И это не случайный недостаток языка.

В этой ситуации уместно использовать ряд возможностей языка C#, известных как средства обработки исключительных ситуаций. Термин «исключительная ситуация» можно считать синонимом понятия «ошибка» со следующей оговоркой – кроме стандартных ошибок (деление на ноль, обращение к несуществующему файлу и т.д.) исключительные ситуации могут описывать боле широкий круг обстоятельств, которые программист считает ошибочными. Таким образом, речь идет о возможности определять свои собственные исключительные ситуации.

В .NET имеется стандартный класс `Exception`, который представляет объект, содержащий информацию о возникшей в ходе выполнения программы, ошибке. Этот объект для стандартных ошибок создается автоматически. Если же Вам нужно создать собственную исключительную ситуацию, то простейшим способом это сделать будет использование оператора следующего вида:

```
throw new Exception("строка с описанием ошибки");
```

Здесь с помощью конструктора класса `Exception` создается объект-ошибка, хранящий указанное параметром строковое описание. В дальнейшем к этой строке можно получить доступ с помощью свойства `Message` класса `Exception`. Далее, оператор `throw` «активизирует» этот объект, что обычно приводит к аварийному завершению программы с выдачей указанного сообщения в стандартном окне, формируемом операционной системой. Таким образом, новая версия метода `SetHeight` выглядит следующим образом:

```
public void SetHeight (double newHeight)
{ if ((newHeight>0)&&( newHeight<230)&&
(newHeight>height))
   height=newHeight;
  else throw new Exception("недопустимая высота");
}
```

Пока что наша программа сумела сгенерировать специфическую для класса `Person` ошибку. Будет еще лучше, если мы научимся обрабатывать такую ошибку. Под обработкой ошибки не следует понимать полную нейтрализацию ошибочной ситуации с выводом программы в нормальный режим работы. Нельзя назвать эвакуацию населения в большом населенном пункте выходом в нормальный режим. Обработка ошибок – это сравнительно небольшие программные действия по ликвидации последствий аварийного завершения программы. Будет сделана попытка предпринять эти действия непосредственно в момент возникновения ошибки, но избежать аварийного, с точки зрения операционной системы, выхода из программы. Программа продолжит свое выполнение с «минимальными» потерям.

Продемонстрируем эти новые языковые возможности на примере метода Main, использующего объекты класса Person:

```
static void Main(string[] args)
{ Person p = new Person();
  double age = Convert.ToDouble(Console.ReadLine());
  try
  { p.SetHeight(age); }
  catch
  { Console.WriteLine("Неверный рост"); }
}
```

Обратите внимание на появление конструкции

```
try
{ блок операторов }
catch
{ блок операторов }
```

Каждый из двух новых блоков (try и catch) может состоять из произвольного количества операторов и формирует самостоятельную область видимости переменных.

Если при выполнении операторов try-блока происходит ошибка, дальнейшее выполнение передается catch-блок. Обратите внимание, что ошибка может произойти в любом месте try-блока. Это может быть встроенная ошибочная ситуация (например, деление на 0) или ошибка, реализованная программистом с помощью оператора throw. В нашем случае ошибка произойдет, если пользователь введет некорректное значение возраста (например, отрицательное число). Если же пользователь введет строку, не являющуюся изображением числа, то программа не сумеет обработать эту ошибку. Дело в том, что такая ошибка происходит при выполнении метода Convert.ToDouble, которое в примере происходит за пределами try-блока. Поэтому переместим соответстующий оператор в try-блок:

```
try
{ double h = Convert.ToDouble(Console.ReadLine());
  p.SetHeight(h);
}
catch
{ Console.WriteLine("Неверный рост"); }
```

Ситуация улучшилась. Но теперь становится актуальной другая проблема – в программе возникают ошибки различных типов. Однако их обработка выполняется одинаково. В нашем случае – выдачей сообщения "Неверный рост". С этим можно справиться, используя параметр в заголовке catch-блока:

```
try
{ double h = Convert.ToDouble(Console.ReadLine());
  p.SetHeight(h);
}
catch (Exception e)
{ if(e.Message=="недопустимый рост")
    Console.WriteLine("Неверный рост");
  else
    Console.WriteLine("Другая ошибка");
}
```

Теперь возникающий во время выполнения объект-ошибка как фактический параметр передается в формальный параметр `e` блока `catch`. В классе `Exception` имеется свойство `Message`, значение которого для объекта-ошибки определяется в момент его создания. В нашем случае мы создаем объект-ошибку со значением Message равным `"недопустимый рост"`. Благодаря этому в `catch`-блоке удается распознать тип ошибки и правильно на нее отреагировать.

Свойства класса

Свойства – это специализированные методы класса, предоставляющие открытый доступ к закрытым переменным класса. Приведем пример свойств, определенных в классе `Person`:

```
public double Height
{ get {return height; }
  set
  { if (value >0)&&( value <230)&& (value >height)
      height=value;
  }
}
```

Использование такого свойства в клиентской части программы очень напоминает использование обычных переменных:

```
Console.WriteLine(p.Height); //используется get-блок для чтения значения
p.Height=178; //используется set-блок записи значения
p.Height++;  //используется get- и set-блоки
```

Перечислим синтаксические особенности свойств:

1. Свойства обычно являются открытыми.
2. Свойства не имеют списка параметров (даже пустого).
3. Тело свойства может состоять из двух блоков операторов – первый начинается со слова `get`, а второй – со слова `set`. Каждый из этих блоков не является обязательным, но хотя бы один из них должен быть определен.

Когда в клиентской части происходит обращение к свойству по чтению, управление передается в `get`-блок, который должен вернуть интересующее нас значение закрытой переменной.

Когда в клиентской части происходит обращение к свойству по записи, управление передается в `set`-блок, который должен обеспечить корректное присваивание закрытой переменной нового значения. Это новое значение представлено в теле `set`-блока ключевым словом `value`, которое играет роль формального параметра, принимающего фактическое значение, указываемое в клиентском коде.

Если в свойстве не определен `set`-блок, то реализуется доступ только по чтению. Если в свойстве не определен `get`-блок, то реализуется доступ только по записи (полезно для паролей). Можно реализовать и более экзотические варианты, например, чтение и однократная запись.

Свойства можно использовать для представления такой информации об объектах класса, которая не представлена непосредственно в виде переменных класса, но может быть вычислена на их основании.

Например, если в классе `Triangle` (треугольник) имеются переменные, соответствующие длинам трех сторон, то можно определить полезное свойство `Perimeter` (периметр). Однако такое свойство будет реализовывать доступ только по чтению:

```
class Triangle
{ private double a,b,c;
  public double Perimeter { get { return (a+b+c); } }
```

Этот пример служит демонстрацией важного преимущества инкапсуляции. Даже если внутреннее представление данных класса изменится, программный код, использующий этот класс, изменять не придется. Допустим, в представлении треугольника вместо трех длин сторон решено использовать две длины сторон и угол между ними. Тогда в классе `Triangle` нужно соответствующим образом изменить реализацию свойства `Perimeter`.

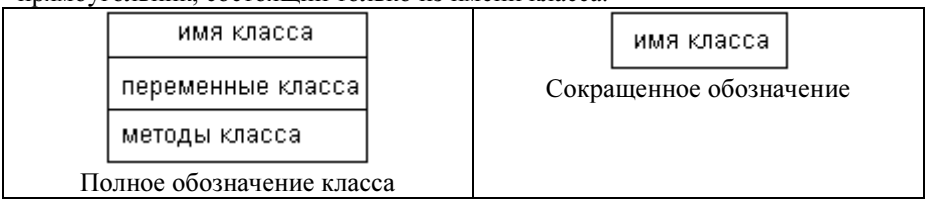
Язык UML

Написание программ – процесс очень сложный. И поэтому важно иметь средства, облегчающие общение между программистами, а также другими участниками программного проекта. Язык программирования для этой цели не подходит – текст программы быстро становится громоздким, и проследить взаимосвязи в нем становится очень трудно. В этом случае обычно используются более наглядные визуальные конструкции.

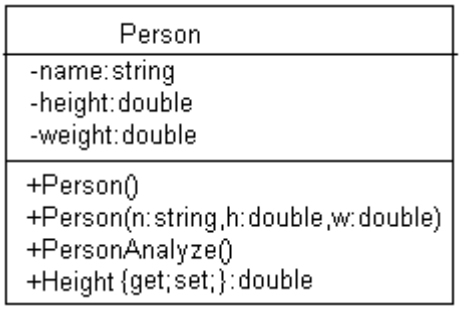
На данный момент сообщество программистов выработало достаточно универсальный и, что еще важнее, стандартный язык таких обозначений – Unified Modelling Language (UML – унифицированный язык моделирования).

В состав этого языка входит довольно большое количество разновидностей обозначений – так называемых диаграмм. В данном кратком пособии будем использовать только одну разновидность диаграмм – статические диаграммы классов (в дальнейшем – диаграммы классов).

На диаграмме классов каждый класс изображается прямоугольником, разделенным на три части. В первой указывается только имя класса, во второй перечисляются переменные класса, в третьей – методы класса. Иногда имеет смысл использовать сокращенное обозначение класса – прямоугольник, состоящий только из имени класса.



Перед открытыми переменными и методами класса указывается знак + (вместо `public`), а перед закрытыми – знак – (вместо `private`). Кроме того, при описании переменных, методов и параметров методов сначала указывается имя, а затем после символа «:» тип. На диаграмме класса реализация метода не указывается. Таким образом, детализированная диаграмма класса `Person` будет выглядеть следующим образом:



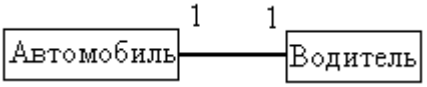
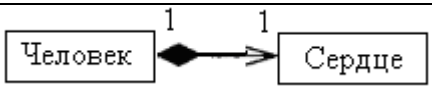
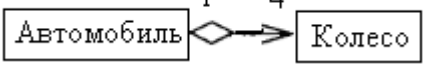
Обратите внимание на описание свойства `Height` – справа от его имени указывается пара фигурных скобок с ключевыми словами `get` и/или `set`.

Связи между объектами

Ранее рассмотренные классы демонстрировали способность объединять в себе несколько переменных различных встроенных примитивных типов (`int`, `double`, `char`, `bool`, `string`). Это позволяет успешно моделировать объекты, которые в процессе своего

функционирования слабо взаимодействуют с другими объектами. Однако в большинстве систем именно такие взаимодействия и представляют наибольший интерес.

В языке UML сделана попытка классифицировать типы связей между объектами. Такая классификация существенно помогает в описании больших программных систем. Кроме того, существуют стандартные приемы реализации того или иного вида связи.

	Описание	Обозначение
Ассоциация	объект связан с другими объектами (знает об их существовании). Автомобиль – водитель. Человек – супруг.	
Композиция	объект (обязательно) состоит из других объектов (подобъектов) Подобъекты не могут существовать без объекта. Человек – сердце. Книга – автор.	
Агрегация	объект (обязательно) состоит из других объектов (подобъектов). Подобъекты могут существовать самостоятельно или находиться в агрегации с другими объектами.	

Таким образом, ассоциация – наиболее «слабый» тип связи, в наименьшей степени регламентирующий особенности связи.

Числа на концах линий связей называются кратностями связи.

Для реализации всех типов связей нужно в одном классе разместить переменную, ссылающуюся на объект (объекты) другого класса.

Для реализации ассоциации следует предусмотреть метод присваивания этой переменной ссылки на объект и метод, прерывающий (обнуляющий) эту связь. В конструкторе эту связь устанавливать не нужно.

Для реализации композиции следует в конструкторе класса создать объект и присвоить ссылку на него переменной. Открытый доступ к этому объекту реализовать только по чтению.

Для реализации агрегации следует в конструктор класса передать готовый объект и присвоить ссылку на него переменной. Реализовать также метод присваивания этой переменной ссылки на другой объект. Важно гарантировать невозможность присваивания этой переменной значения `null`.

Наследование (Inheritance)

Наследование – второй важнейший принцип ООП (после инкапсуляции). Он заключается в создании новых классов, расширяющих возможности уже имеющихся. Допустим, к этому моменту Вы располагаете достаточно функциональным классом `Person`, позволяющим успешно программно моделировать различные ситуации из мира людей. На следующем этапе Вы поняли, что многие последующие задачи будут использовать в качестве объектов студентов. Естественно, следует разработать класс `Student`. Однако понимание того что «студент является человеком» (то есть «человек» - общее понятие, а «студент» - частное), подсказывает, что создавать класс `Student` опять «с нуля» не разумно. Некоторую часть информации и возможностей студент «наследует» у человека. Существует два способа реализации наследования: а) классическое (реализует отношение «is_a») и б) модель делегирования внутреннему члену (отношение «has-a»). Наследование обеспечивает возможность повторного использования программного кода.

В дальнейших примерах будем использовать несколько измененный класс `Person`:

```
class Person
{ private string name; //protected!!!
  private List<Person> acq; // список знакомых
  public Person(string n)
  { name = n; acq = new List<Person>(); }
  public string Name { get {return name;}}
  public void GetAcq(Person p) // познакомиться
  { if (!acq.Contains(p)) acq.Add(p); }
  public void UnGetAcq(Person p) //разорвать знакомство
  { if (acq.Contains(p)) acq.Remove(p); }
  public string Greeting(Person p)
```

```
  { if (acq.Contains(p))
      return String.Format("Hi, {0}!", p.Name);
    else return String.Format("Hello, Mr. {0}!",
p.Name);
  }
}
```

Такой класс `Person` кроме имени, снабжает каждого человека множеством знакомых, и соответствующими возможностями знакомиться и прерывать знакомство. Заметим, что реализовано не обоюдное знакомство (поробуйте исправить это самостоятельно). Поведение класса `Person` можно протестировать следующим образом:

```
Person p1 = new Person("John");
Person p2 = new Person("Ann");
Console.WriteLine(p1.Greeting(p2));
p1.GetAcq(p2);
Console.WriteLine(p1.Greeting(p2));
Console.WriteLine(p2.Greeting(p1));
p1.UnGetAcq(p2);
Console.WriteLine(p1.Greeting(p2));
```

Допустим, Вам нужно реализовать программу, моделирующую некоторые черты поведения студентов. На данный момент класс `Person` мало приспособлен для реализации таких задач. Реализуем следующий класс `Student`:

```
class Student:Person
{ private int year;  //год обучения
  private List<String> courses; //список изучаемых
курсов
  public Student(string n,int year):base(n)
  {this.year=year; courses=new List<String>();}
  public void GetCourse(String c) //студент выбирает
курс
  { if (!courses.Contains(c)) courses.Add(c); }
  public string SayCourses()
  { string s =
      String.Format("{0}.изучает следующие
курсы:\n",Name);
    foreach(String c in courses) s+=c+'\n';
    return s;
  }
}
```

Самое важное здесь находится в заголовке класса.

```
class Student:Person
```

Такая конструкция обозначает, что класс `Student` наследует от класса `Person` все его переменные и методы (кроме конструктора!). Таким образом в классе `Student` кроме переменных `year` и `courses` неявно присутствуют переменные `name` и `acq`. Аналогично, кроме методов `GetCourse` и `SayCourses`, в классе неявно присутствуют методы `GetAcq`, `UnGetAcq`, `Greeting` и свойство `Name`.

Как уже было сказано, конструкторы не наследуются. Поэтому у класса Student только один конструктор.

Класс Person	Класс Student	
	Унаследованные элементы	Собственные элементы
Переменные		
`name`	`name`	`year`
`acq`	`acq`	`cources`
Методы		
Конструктор `Person`		Конструктор `Student`
Свойство `Name`	Свойство `Name`	
Метод `GetAcq`	Метод `GetAcq`	Метод `GetCourse`
Метод `UnGetAcq`	Метод `UnGetAcq`	Метод `SayCourses`
Метод `Greeting`	Метод `Greeting`	

Отношение наследования описывается несколькими синонимичными терминами. Когда класс B является наследником класса A, то класс A называют базовым, а класс B - производным. Иногда используют другие термины: предок и потомок или суперкласс и подкласс.

Теперь главный вопрос – зачем нужно наследование? По мере Вашего программистского опыта Вы будете находить все новые ответы на этот вопрос. Сейчас ограничимся следующими:

1. Наследование увеличивает степень повторного использования программного кода. Очевидно, что текст класса Student выглядит довольно компактно, по сравнению с его действительным содержимым.
2. Наследование способствует конструированию программного кода путем использования абстракции и специализации. Многие мыслительные процессы существенно опираются на оперирование общими и частными понятиями. Это помогает описывать мир на естественном языке. В программировании такая наследование позволяет сохранить такой стиль мышления, а следовательно и программного моделирования.
3. Наследование является основой полиморфизма. Объяснение этой причины будет дано несколько позже.

Пример `Main`, где используются базовые и производные возможности.

```
1  Person p1 = new Person("John");
```

```
2  Student s1 = new Student("Vasya", 2);
3  Student s2 = new Student("Kolya", 2);
4  s1.GetAcq(s2);
5  s1.GetAcq(p1);
6  s1.GetCourse("ООП");
7  s1.GetCourse("БД");
8  s1.SayCourses();
9  p1.GetAcq(s1);
```

В строке 4 объектом `s1` вызывается унаследованный метод `GetAcq`. Тип его формального параметра – `Person`. Однако в качестве фактического значения передается переменная типа `Student`. Это не является ошибкой. Здесь действует следующее правило совместимости типов:

Переменным базового класса можно присваивать ссылки на объекты производных классов.

Это правило имеет интуитивно понятное объяснение – студент является частным случаем человека и, поэтому, для него допустимо то, что допустимо для человека.

Аналогичные рассуждения действуют и для строки 9.

Класс Object

Язык C# в существенной степени является объектно-ориентированным языком программирования. Все типы языка принадлежат к одной из следующих категорий:

1. Перечислимые типы.
2. Структурные типы.
3. Классы.
4. Интерфейсы.
5. Делегаты.

Перечислимые и структурные типы играют в языке сравнительно скромную роль. С интерфейсами и делегатами Вы познакомитесь позже. Большинство типов гигантской библиотеки базовых классов .NET являются классами. Даже примитивные встроенные типы реализованы с помощью классов (`int` – с помощью `Int32` и т.д.). И что самое удивительное – у всех этих тысяч классов имеется общий базовый класс, который называется `Object`. Любой Ваш собственный класс имеет неявный базовый класс `Object`, а вместе с ним и несколько унаследованных членов. На данный момент трудно во всех подробностях объяснить преимущества такого подхода. Однако простой аргумент очевиден уже сейчас – класс `Object` является отправной точкой всей системы классов. А поскольку в C# не поддерживается множественное наследование (класс не может быть

наследником нескольких из базовых классов), вся система классов имеет аккуратную древовидную структуру.

Защищенные переменные

В предыдущем примере производный класс, выполняя свой метод `SayCourses`, получает доступ к значению переменной `name` с помощью свойства `Name`. Это кажется немного странно, поскольку переменная `name` унаследована классом `Student` от класса `Person` и, таким образом, входит в состав каждого объекта класса `Student`. Однако, поскольку в классе `Person` переменная `name` описана как закрытая (`private`), она недоступна даже внутри своего производного класса `Student`.

Эта ситуация встречается часто при использовании производных классов, что ухудшает производительность программы за счет «лишних» вызовов методов и свойств. Для преодоления этой проблемы в языке C# имеется еще один уровень защиты переменных, обозначаемый ключевым словом `protected`. Такие «защищенные» переменные становятся непосредственно доступны не только в своем классе, но и во всех производных от него.

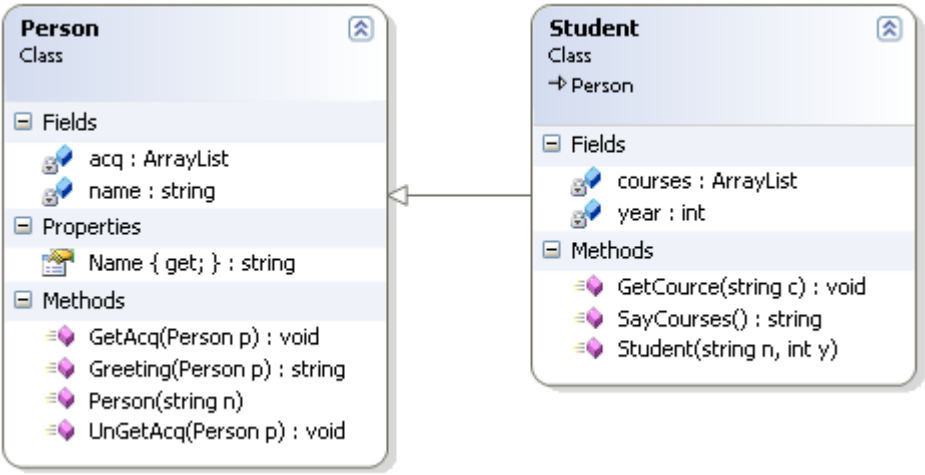
Если в классе `Person` переменную `name` описать как защищенную:

```
protected string name;
```

в реализации метода `SayCourses` можно напрямую обращаться к переменной:

```
String.Format("{0}.изучает следующие курсы:\n",Name);
```

Наследование нашло свое наглядное отражение и на диаграммах классов UML. Базовый и производный класс соединяются стрелкой с треугольным наконечником как показано на следующей диаграмме:



Обратите внимание, что здесь используется нестандартный вариант диаграммы классов, используемый только в рамках средств разработки Microsoft. На таких диаграммах тело класса, кроме переменных (Fields) и методов (Methods), может содержать отдельный перечень свойств (Properties). Кроме того, перед каждым членом класса указывается некоторый значок, который заменяет собой модификатор доступа и указывает на категорию элемента класса.

Стандартный UML имеет большее распространение и позволяет разработчикам эффективно общаться, независимо от используемых средств разработки.

Вызов базового конструктора

Вернемся к конструктору класса `Student`:

```
public Student(string n,int year):base(n)
{this.year=year; courses=new List<String>();}
```

Что означает конструкция **`:base(n)`** в заголовке? Дело в том, что конструирование объекта производного класса подразумевает инициализацию унаследованных переменных. С этой работой успешно справляется конструктор производного класса. Однако воспользоваться им в конструкторе производного класса непосредственно не удастся по двум причинам:

1. Конструкторы вообще нельзя вызывать как обычные методы.
2. Конструкторы не наследуются.

Единственным способом заставить работать конструктор базового класса и является конструкция `base()` в заголовке конструктора производного класса.

В примере видно, что один из параметров – n – используется при вызове `base(n)`, а второй – непосредственно в теле конструктора.

Вызов базового конструктора Вам не понадобится в той редкой ситуации, когда производный объект инициализирует унаследованные переменные не таким образом, как базовый конструктор.

Переопределение методов. Обращение к «затененным» элементам класса

Сейчас в базовом классе `Person` имеется метод `Greeting`, возвращающий строку приветствия человека. Этим методом могут пользоваться объекты-студенты, поскольку он унаследован. Однако в реальной жизни молодые люди выполняют свои действия с определенными особенностями. Например, в строке приветствия они могут использовать некоторые «украшения», например вместо «Hi, Peter!" - «Hi, Peter! Ну чё?"

Получается, что базовая реализация метода `Greeting` нас уже не устраивает и мы переопределим в классе `Student` метод `Greeting` точно с таким заголовком (сигнатурой):

```
public string Greeting(Person p)
{ return Greeting(p) + " Ну чё?"; }
```

Идея понятна – склеить строку базового приветствия с «украшением». Однако наша программа после запуска и некоторого раздумья выдает ошибку:

```
Process is terminated due to StackOverflowException.
```

Дело в том, что сейчас метод `Greeting` рекурсивно вызывает сам себя и этот процесс бесконечен. Случилось это потому, что новый метод `Greeting` с той же сигнатурой, что и базовый «заслонил» собой базовый. Это не означает, что базовый метод исчез. Однако для его вызова опять придется использовать ключевое слово `base`:

```
public string Greeting(Person p)
{ return base.Greeting(p) + " Ну чё?"; }
```

Ключевое слово base в данном случае указывает на то, что метод Greeting нужно вызывать из базового класса.

Переопределение методов – очень полезный прием. Но наибольшую пользу он принесет только в форме полиморфного поведения, о чем речь пойдет дальше.

Аналогично можно осуществить и переопределение переменных. Однако это прием используется довольно редко, поскольку быстро вносит неразбериху в ассортимент переменных производных классов.

```
class Program
{ static void Main(string[] args)
  { A a = new A(); Console.WriteLine(a.x);
    B b = new B(); Console.WriteLine(b.x);
Console.WriteLine(b.oldX);
  }
}
class A   {   public int x=45;   }
class B : A
{ public bool x=true;
  public B() { base.x = 34; }
  public int oldX { get { return base.x; } }
}
```

Многоуровневое наследование

Многоуровневое наследование подразумевает возможность использовать производный класс в качестве базового для определения еще одного производного класса. Например, класс Student может стать основой для создания класса студент-магистр, который должен написать магистерскую работу:

```
class Magister : Student
{ private string diploma;
  public Magister(string name, int year, string
diploma)
    : base(name, year)
  { this.diploma=diploma; }
        . . .
}
```

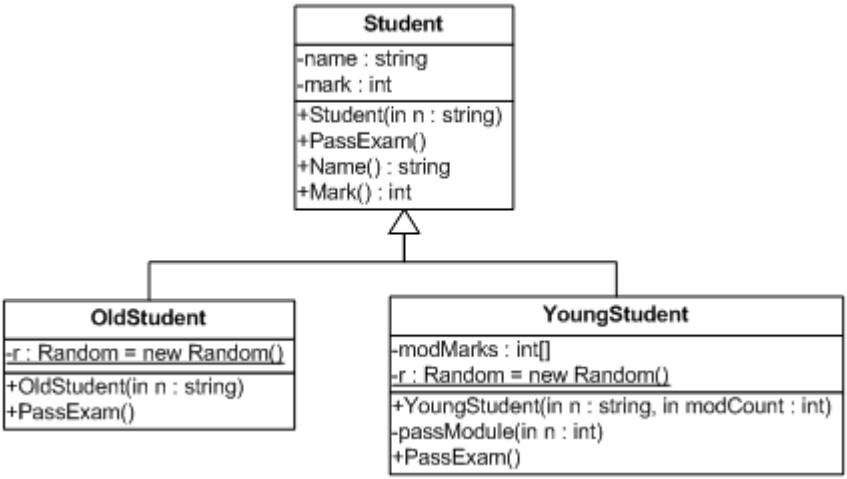
Полиморфизм

Использование наследования часто приводит к созданию нескольких производных от данного базового классов. Например, мы можем определить несколько производных классов от класса `Student`. Важной способностью любого студента является способность сдавать экзамен. Допустим, что студенты младших курсов сдают экзамен путем сдачи серии учебных модулей, в результате чего окончательная оценка накапливается как результат оценок по каждому модулю. Студенты старшекурсники сдают «классический» экзамен – в конце учебного периода. Несмотря на разницу в способах сдачи экзаменов, во многих ситуациях управления студентами не хотелось бы постоянно учитывать описанное различие. Приведем программную реализацию данной ситуации.

```
class Student
```

```
{ private string name;
  private int mark;
  public Student(string n) { name = n; }
  public void PassExam() { mark = 0; }
  public string Name { get { return name; } }
  public int Mark
  { get { return mark; } set { mark = value; } }
}
class YoungStudent : Student
{ private int[] modMarks;
  private static Random r=new Random();
  public YoungStudent(string n, int modCount)
    : base(n)
  { modMarks = new int[modCount]; }
  private void passModule(int n)
  { modMarks[n] = r.Next(0, 13); }
  public void PassExam()
  { double s = 0.0; double d;
    for (int i = 0; i < modMarks.Length; i++)
    { passModule(i); s += modMarks[i]; }
    d = s / modMarks.Length;
    Mark = (int)(Math.Round(d));
  }
}
class OldStudent : Student
{ private static Random r=new Random();
  public OldStudent(string n) : base(n) {  }
  public void PassExam()
  { Mark = r.Next(0, 13); }
}
```

UML-диаграмма трех «студенческих» классов:



Класс `Student` сам по себе для создания объектов использоваться не будет. Поэтому он содержит сугубо формальную реализацию метода `PassExam`. Однако другие его методы будут успешно использоваться объектами производных классов без переопределения.

Далее в клиентской части (класс `Program`) мы решаем следующие задачи:

1. Создание множества студентов. Студенты двух типов помещаются в `ArrayList`.
2. Все созданные студенты сдают экзамен.
3. Выводится статистика оценок по всем студентам.

```
class Program
{ private static ArrayList students;
  static void MakeStudents()
  { students = new ArrayList();
    students.Add(new YoungStudent("Peter",2));
    students.Add(new OldStudent("Terry"));
    students.Add(new YoungStudent("Frank",2));
    students.Add(new OldStudent("Ann"));
  }
  static void PassExams()
  { foreach (Student student in students)
    { switch (student.GetType().Name)
      { case "YoungStudent":
        { ((YoungStudent)student).PassExam(); break; }
        case "OldStudent":
        { ((OldStudent)student).PassExam(); break; }
```

```
      }
    }
  }
  static void Report()
  { foreach (Student st in students)
      Console.WriteLine("Student {0} has mark {1}",
        st.Name, st.Mark);
  }
  static void Main(string[] args)
  { MakeStudents(); PassExams(); Report(); }
}
```

Ключевым моментом этой программы является реализация метода `PassExam`. Идеальным по простоте был бы следующий его вариант:

```
static void PassExams()
{ foreach (Student student in students)
student.PassExam(); }
```

К сожалению, в таком случае все студенты получили бы оценку 0, поскольку для всех студентов в этом случае работает метод `PassExam` из базового класса. Дело в том, что решение о том, какой вариант метода `PassExam` вызывать принимается на стадии компиляции (это называется ранним связыванием) на основании типа объекта. А в приведенной реализации переменная цикла `student` описана базовым классом `Student`.

В результате приходится использовать «тяжелую артиллерию» языка C# - средства для работы с информацией о типах во время выполнения программы (Run Time Type Information – RTTI). Основным средством этой категории является метод `GetType`, возвращающий информацию о типе объекта, на который ссылается переменная во время выполнения программы. Конструкция `student.GetType().Name` возвращает строку с именем этого типа. Такое решение не только громоздко, но и не надежно. Представьте, как тяжело обудет поддерживать правильность такого программного кода, если:

1. Будут возникать все новые производные классы студентов.
2. Подобные методы, основанные на switch-анализе вариантов, встречаются во многих местах программы.

Данная проблема является типичной для объектно-ориентированного способа разработки программ и имеет свое решение. Разработчикам языка C# (и других ОО языков) удалось предложить средства, обеспечивающие позднее связывание. , которое и известно под названием полиморфизма.

Полиморфная реализация метода `PassExam` потребует следующих шагов:

1. В базовом классе описать метод с ключевым словом `virtual`:

```
class Student
{ . . .
  public virtual void PassExam() { mark=0;}
  . . .
}
```

2. В производных классах описать метод с ключевым словом override:

```
class YoungStudent : Student
{ . . .
 public override void PassExam() { . . .}
  . .
}
class OldStudent : Student
{ . . .
 public override void PassExam() { . . .}
  . .
}
```

Вот и все! Теперь метод PassExam идеально прост:

```
static void PassExams()
{ foreach (Student st in students) st.PassExam();  }
```

Теперь решение о том, какой из методов PassExam вызывать откладывается на стадию выполнения программы и зависит не от «статического» типа переменной student, а от ее «динамического» типа, то есть от реального типа объекта, на который ссылается переменная в тот или иной момент выполнения программы.

Термин «полиморфизм» (в буквальном переводе с греческого – многоформенность) означает возможность разнотипных объектов самостоятельно продемонстрировать различие в своем поведении без выяснения типа этих объектов извне.

Метод ToString

Характерным примером полиморфного метода является метод ToString. Он определен в классе Object как virtual. Поэтому в производных классах такой же метод удобно переопределять как override. Это обеспечит однотипное (полиморфное) управление Вашими объектами. Например, в классе Student его реализация может быть следующей:

```
public override string ToString()
{return String.Format("Student {0} has mark
{1}",name,mark);}
```

Благодаря этому упростится реализация метода Report:

```
static void Report()
```

```
{ foreach (Student st in students)
Console.WriteLine(st); }
```

Дело в том, что метод `WriteLine` неявно вызывает для каждого выводимого им объекта его мето `ToString`.

Типичные ситуации проявления полиморфизма

Сначала определим примитивные классы:

```
class A
{ public virtual void Do() { Console.Write("A works ");
} }
class B:A
{ public override void Do() { Console.Write("B works
"); } }
```

Присваивание

```
A w1 = new B(); w1.Do();
```

Параметры метода

```
static void HardWork(A worker)
{ worker.Do(); worker.Do();
  Console.WriteLine("Help!");
  worker.Do(); worker.Do();
}
```

Затем в Main:

```
HardWork(new B());
```

Результат:

B works B works Help! B works B works

Таким образом, метод HardWork успешно сочетает два вида действий:
- действия, независимые от типа передаваемого параметра (“Help!”);
- действия, полиморфно зависимые от типа передаваемого параметра.

Абстрактные классы и полиморфизм

Нередко бывают ситуации, когда следует гарантировать невозможность создания объектов класса. В частности, это бывает для базовых классов, соответствующих абстрактным понятиям. В таком случае базовый класс следует сделать абстрактным, указав в его заголовке ключевое слово `abstract`. Теперь попытка создания объектом будет распознаваться компилятором как ошибка.

Другая особенность абстрактных классов – наличие в них абстрактных методов. Такие методы имеют две синтаксические особенности:
1. В заголовке абстрактного метода указывается ключевое слово `abstract`.
2. Абстрактный метод не имеет реализации.

Теперь можно определить базовый класс `Student`:

```
public abstract class Student
{ private string name;
  private int mark;
  public Student(string n){name=n;}
  public abstract void PassExam();
  public string Name { get { return name; } }
  public int Mark
  { get { return mark; } set { mark = value; } }
}
```

Наличие абстрактного метода позволяет реализовать принудительный полиморфизм. Дело в том, что производный класс обязан реализовать абстрактный метод (иначе этот класс остается абстрактным и должен быть описан как `abstract`). Абстрактные методы по умолчанию считаются описанными как `virtual` (указывать `virtual` даже нельзя).

ЛИТЕРАТУРА

1. Троелсен Э. Язык программирования C# 2005 и платформа .NET 2.0 3-е издание.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2007. – 1168 с.: ил.
2. Лабор В.В. Си Шарп: Создание приложений для Windows – Мн.: Харвест, 2003.-384с.
3. Микелсен К. Язык программирования C#. Лекции и упражнения. Учебник: Пер. с англ./ - СПб.: ООО «ДиаСофтЮП», 2002. – 656 с.

Учебное издание

Пенко Валерий Георгиевич, Пенко Елена Анатольевна

Программное обеспечение ЭВМ. Часть 1
Методическое пособие для студентов отделения прикладной математики и факультета информационных технологий

Издано в авторской редакции

Підп. до друку 28.05.2010. Формат 60х90/16.
Гарн. Таймс. Тираж 50 прим.

Редакційно-видавничий Центр
Одеського національного університету
імені І.І. Мечникова,
65082, м. Одеса, вул. Єлісаветинська, 12, Україна
Тел.: (048) 723 28 39