Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Архитектуры программных систем

1. Сравнение ОО языков C#, Java и C++.

2. Язык C# и программирование на платформе .NET

Проф. Забудский Е.И.

Москва 2008

Приводится описание основных возможностей ОО языка программирования C#, а также его сравнение с ближайшими родственниками: языками Java и C++ [1, 4]

В тексте даны ссылки на термины [2, см. с.2 У_М_Комплекса]	
Терминология дисциплины (ч. 1)	Определения основных терминов платформы **.NET Framework**
Терминология дисциплины (ч. 2)	Объектно-ориентированный **а**нализ (ОО**A**) и объектно-ориентированное **п**роектирование (ОО**D**)
Терминология дисциплины (ч. 3)	Ключевые слова языка **C# и их семантика**

Уважаемые студенты!

Основная цель, которую необходимо достигнуть в результате изучения дисциплины Объектно-ориентированный анализ и программирование – научиться разрабатывать компьютерные модели реальных и концептуальных систем соответствующих направлению Бизнес-информатика.

Необходимым условием усвоения дисциплины является
ВАША самостоятельная работа

// **КОММЕНТАРИЙ**. На сайте http://www.firststeps.ru/ “Первые шаги” представлено много интересных обучающих материалов по различным интегрированным средам и языкам программирования, в том числе представлены **C#** и платформа **.NET** (step by step).

Введение

В **июне 2000 года** компания Microsoft анонсировала платформу **.NET** и новый язык программирования, получивший название **C#** (**CSharp**) (читается "**Си шарп**").

C# - это строго типизированный объектно-ориентированный язык, призванный обеспечить оптимальное сочетание удобства, простоты, выразительности и производительности. Платформа **.NET** основана на использовании общей среды выполнения кода **CLR** (**Common Language Runtime – см. Терминология дисциплины, ч.1, термин 16**), подобной виртуальной **Java**-машине, и набора библиотек, доступных для ряда языков программирования. Независимо от того, какой язык используется в процессе разработки, программа компилируется в промежуточный код **IL** (**Intermediate Language**).

C# и **.NET** в некотором смысле являются "родственными душами": некоторые свойства языка специально разрабатывались, чтобы обеспечить комфортную работу в среде **.NET**, в то же время и некоторые свойства **.NET** специально закладывались для поддержки **C#**, хотя **.NET** нацелена на использование многих языков программирования. В данном материале в основном рассматривается **C#**, реже платформа **.NET**.

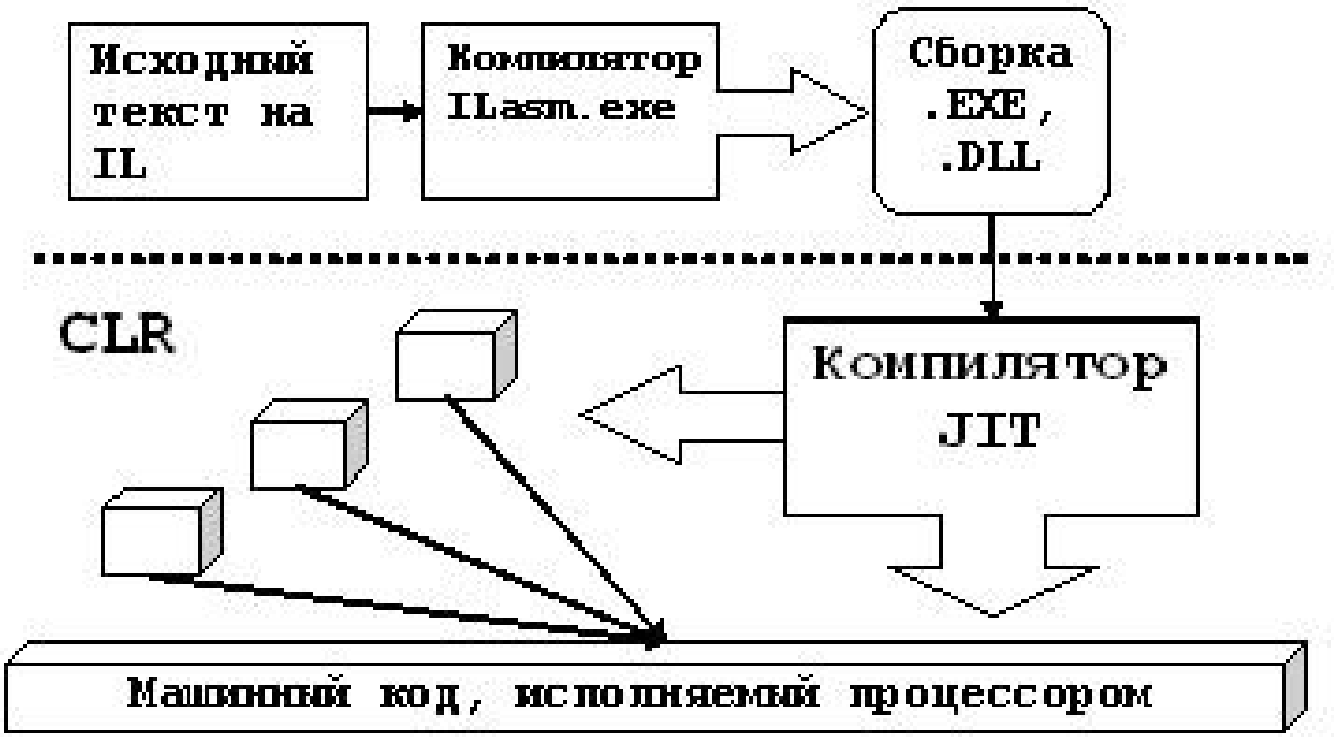

Рис. В.1. Процесс создания и исполнения программы на IL в среде CLR

Одной из особенностей платформы **.NET** является ее «многоязычность». С самых первых шагов по продвижению **.NET**, начиная с лета 2000 года, компания **Microsoft** представляла возможность программирования в среде **.NET** на нескольких языках как очень важную и новую. И это действительно так.

Однако, как отмечено выше, существует один промежуточный язык – один на всех.

На самом деле существует один-единственный язык, на котором создаются программы в среде **.NET** – **IL** (**Intermediate Language – см. Терминология дисциплины, ч.1, тер-**

мин 14) или **MSIL** (**M**icro**S**oft **I**ntermediate **L**anguage). На этом языке можно программировать напрямую (возможность *не означает* необходимость – ***NB***), используя текстовый редактор и компилятор **ILasm.exe**, поставляемый в составе **.NET Framework** (**см. Терминология дисциплины, ч.2, термин 59**) или, используя средства **Visual Studio.NET**.

ASCII-текст программы на языке **MSIL** принято сохранять в файле с расширением **.IL**. Для получения исполняемого файла (**.EXE**) или библиотеки (**.DLL**) надо в командной строке запустить компилятор, указав в качестве параметра имя исходного файла: **ILASM.EXE *HELLO.IL*** /см. выше **рис. B.1**/

В результате будет создан **исполняемый** (**по умолчанию**) или **библиотечный** файл (в зависимости от ключей **/EXE** или **/DLL**), называемый сборкой (**assembly – см. Терминология дисциплины, ч.1, термин 4**). Этот файл представляет собой стандартный программный файл **Windows**, содержащий в заголовке специальную информацию, из которой становится известно, что он должен исполняться не операционной системой, а средой **.NET**, точнее ее исполнительной частью **CLR** (**C**ommon **L**anguage **R**untime).

Сборка не содержит исполняемых машинных команд, которые понимает процессор. Она представляет собой странную смесь из символьных *метаданных* (**см. Терминология дисциплины, ч.1, термин 46**) и собственно команд **IL**, которые опытному программисту напомнят ассемблерные команды (см. выше **рис. B.1**). Во время исполнения **сборки CLR** компилирует код **IL** сборки в машинный, «родной» код процессора, который и исполняет команды. Компиляция сборки в машинный код происходит «по-умному» - компилятор помнит, какие фрагменты кода уже обрабатывались, так что повторной дорогостоящей компиляции не происходит, это делается только тогда, когда нужно. Этот компилятор, являющийся частью **CLR**, так и назыается: **JIT-compiler** (**Just-In-Time – см. Терминология дисциплины, ч.1, термины 35, 36, 37**). Процесс создания и исполнения программы на **IL** в среде **CLR** показан выше **рис. B.1**.

Но к рассмотреному способу создания программ, программисты будут прибегать *крайне редко*.

Код на языке **IL** будет создаваться компиляторами языков **высокого** уровня (один из них компилятор языка **C#**), поставляемых **Microsoft** и ее партнерами.

Язык **C#** разрабатывался с учетом опыта многих предшествующих ему языков программирования, но в первую очередь, конечно, **C++** и **Java**. Родителями **C#** стали Anders Hejlsberg, получивший известность, как автор Delphi, и Scott Wiltamuth.

1. Сравнение ОО языков C#, Java и C++

1.1.A. C# и Java

Ниже приведен перечень свойств, характерных как для **C#**, так и для **Java**. Оба эти языка можно рассматривать как попытку усовершенствовать **C++**, и нужно признать, что в обоих случаях это удалось. **Как можно увидеть из приведенного ниже списка, во многом C# и Java схожи**, но было бы неверно отождествлять эти языки.

A. Исходный текст программы компилируется в промежуточный код, не зависящий от языка и платформы; этот код в дальнейшем выполняется в специальной управляемой среде.

B. Автоматический сбор мусора (**Garbage Collection – см. Терминология дисциплины, ч.1, термин 25**) и запрет на использование указателей. В **C#** допускается ограниченное использование указателей в блоках кода, помечаемых как "ненадежные" (**unsafe – см. Терминология дисциплины, ч.3, термин 71**) .

C. Отсутствие заголовочных файлов. Весь код помещается в пакеты (**packages**) и сборки (**assemblies**). Никаких проблем с порядком объявления классов в случае наличия перекрестных ссылок.

D. **Объекты** (**см. Терминология дисциплины, ч.2, термин 32**) создаются с помощью ключевого слова **new** (**см. Терминология дисциплины, ч.3, термин 41**), выделение памяти производится из "кучи" (**heap**), находящейся в распоряжении среды выполнения.

E. Многопоточность поддерживается путем блокирования объектов

F. Интерфейсы, множественная реализация интерфейсов классом, однократное наследование базового класса производным.

G. Внутренние классы

H. Отсутствие концепции наследования классов с заданным уровнем доступа.

I. Отсутствие глобальных функций и констант, **все элементы должны принадлежать классам.**

J. Массивы и строки со встроенной длиной и проверкой границ.

K. Не применяются операторы «**->**» , «**::**». Во всех случаях используется оператор «**.**».

L. **null** и **boolean** / **bool** являются ключевыми словами.

M. Любая величина должна быть проинициализирована до того, как будет использована.

N. Нельзя использовать целые числа (**integers**) для управления операторами **if**.

O. Блоки **try** могут иметь заключительное предложение **finally** (**см. Терминология дисциплины, ч.3, термины 66, 25**).

1.1.B. C# и C++

Для эффективной работы в среде **CLR** и повышения производительности труда программистов в язык **C#** внесено значительное количество упрощений по сравнению с языком **C++**. Некоторые из них основаны на следующих особенностях:

- исключение отдельного заголовочного файла и препроцессора;
- устранение проблем с управлением памятью за счет: **1)** использования ссылок вместо указателей и **2)** сборки мусора во время исполнения.

Вместе с тем синтаксис **C#** основан на синтаксисе **C++** с небольшими изменениям, которые применяются либо для предоставления новых преимуществ, либо для совместимости со средой исполнения **CLR**.

1.2. Свойства (properties) /см. на с. 28 раздел 2.2.1/ (см. Листинг 2.3 на с.35)

Причина появления свойств в **C#** заключалась в попытке формализовать на уровне синтаксиса языка концепцию методов **get / set**, активно используемых программистами, особенно в инструментах класса **RAD** (**Rapid Application Development**).

Вот так выглядит типичный фрагмент кода, написанный на **C++** или **Java**:

```
foo.setSize(getSize() + 1);
label.getFont().setBold(true);
```

Теперь напишем то же самое на **C#**:

```
foo.size++;
label.font.bold = true;
```

Очевидно, что код C# легче читается и понятнее для тех, кто будет использовать объекты foo и label. Аналогичное **преимущество наблюдается и при реализации свойств**.

Java / C++:

```
public int getSize()              // метод-аксессор
{
        return size;
}
public void setSize (int value)  // метод-мутатор
{
        size = value;
}
```

C#:

```
public int Size                   // свойство [3, C.449, C.541…553]
{
        get                       // get-блок
        {
```

```
                return size;
        }
        set                     // set-блок
        {
                size = value;
        }
}
```

C# предлагает более прозрачный способ реализации свойств, что особенно очевидно для свойств, допускающих чтение и запись. Связь методов **get** и **set** в **C#** становится врожденной, в то время как в **C++** и **Java** она лишь поддерживается. У такого подхода есть много преимуществ. Он заставляет программистов мыслить в терминах свойств, независимо от того доступно ли свойство как для чтения, так и для записи, или оно предполагает только чтение. Если необходимо изменить название свойства, то достаточно будет сделать это в одном месте (а часто методы **get** и **set** оказываются разделенными сотнями строк кода). Комментарии также достаточно ввести в одном месте, так что они никогда не окажутся рассогласованными.

> Можно возразить, что предлагаемый **C#** синтаксис не дает реальных преимуществ, так как в случае его использования нельзя с уверенностью сказать, с чем мы имеем дело, с **полем** (**см. Терминология дисциплины, ч.2, термин 42**) или **свойством**. Но практически никогда реальные классы, спроектированные на **Java** (и естественно на **C#**), не имеют общедоступных (**public**) полей. Поля обычно имеют ограниченный уровень доступа (**private / protected**) и раскрываются только через функции **get / set**, где **C#** как раз и предлагает более удобный синтаксис. Кроме того очевидно, что если класс правильно спроектирован, **то пользователя должна интересовать только спецификация класса**, а **отнюдь не его реализация**.
>
> Еще один аргумент противников использования свойств - снижение эффективности кода. Однако хороший компилятор может свести реализацию простого метода получения значения поля (**get**) к **in-line** функции, что сделает ее выполнение столь же быстрым, как и непосредственное считывание значения поля.

1.3. Индексаторы (indexers) /см. на с. 29 раздел 2.2.2/ (см. Листинг 2.3 на с.35)

В **C#** предусмотрены средства для создания пользовательских классов-контейнеров, к внутренним элементам которых можно обращаться с помощью того же оператора индекса, что и к элементам обычного массива встроенных типов. Метод, который обеспечивает такую возможность, получил название **индексатор** (**indexer**). Индексатор представляет собой слегка измененное свойство **C#**, в простейшем случае индексатор создается через синтаксическую конструкцию **this[ ]**, например **[3, С.449, С. 553…562]**:

```
public class Skyscraper
{
   Story[ ] stories;
   public Story this[int index]
```

```
    {
        get
        {
                return stories[index];
        }
        set
        {
                if (value != null)
        {
                stories[index] = value;
        }
        }
    }
}
```

Skyscraper empireState = **new** Skyscraper (...);

empireState[102] = **new** Story("The Top One", ...);

За исключением ключевого слова **this** индексатор ничем не отличается от обычного объявления свойства **C# [3, С.553-562]**.

См. Терминология дисциплины, ч.3, термин 63

см. Терминология дисциплины, ч.1, термин 73

1.4. Делегаты (delegates) /см. на с. 30 раздел 2.2.5/ [3, С.729-740]

Понятие **делегат** (**delegate**) в **C#** можно рассматривать как безопасный объектно-ориентированный указатель на функцию.

Делегат в **C#** выполняет те же действия, что и указатель на функцию в **C++** или интерфейс в **Java**.

По сравнению с указателями на функции **делегаты являются более безопасными и могут поддерживать много методов.**

Делегаты выигрывают и у интерфейсов **Java**, так как для вызова метода в **C#** не нужно определять внутренние адаптеры или дополнительный код для вызова нескольких методов.

Наиболее важная роль делегатов состоит в том, что они используются для обработки событий, как показано в следующем **разделе 1.5**.

См. Терминология дисциплины, ч.1, термин 19

См. Терминология дисциплины, ч.3, термин 16

См. Терминология дисциплины, ч.2, термин 8

1.5. События (events) /см. на с. 31 раздел 2.2.6/ [3, С. 740-749]

C# обеспечивает непосредственную поддержку событий.

Казалось бы, обработка событий должна быть фундаментальной частью программирования, однако большинство языков программирования уделяет формализации этой концепции неожиданно **мало внимания**.

Если попытаться проанализировать существующие на данный момент подходы к обработке событий, то можно выделить указатели на функции в **Delphi**, внутренние адаптеры в **Java**, и, конечно, системы сообщений **Windows API**.

Далее в **Листинге 1.1** показано, как в C# выглядит процесс: **1)** описания, **2)** генерации и **3)** обработки события:

```
1  // Объявление делегата определяет сигнатуру метода, который может быть вызван
2  public delegate void ScoreChangeEventHandler(int newScore, ref bool cancel);          // 1)

3  // Класс-делегат  Game, генерирующий событие : см. строку 20 ;                        2)
4  public class Game
5  {       /* Обратите внимание на использование ключевого слова event;
                              ScoreChange – событие: см. строки 6 и 20 */
6     public event ScoreChangeEventHandler ScoreChange;
7     int score;                                                  // переменная score –  счет
8                                                                 // Свойство Score – Счет
9     public int Score
10    {
11       get
12       {
13          return score;
14       }
15       set
16       {
17          if (score != value)
18                {
19                      bool cancel = false;
20                    ScoreChange (value, ref cancel);
21                if (! cancel) score = value;
22                }
23       }
24    }
25 }                          // Окончание класса Game

26 // Класс Referee - обработчик события                                                  3)
27 public class Referee
28 {
29    public Referee (Game game)
30    {
```

```
31  /* Следит за изменением счета игры: событие ScoreChange инкапсулирует метод-
    обработчик game_ScoreChange, который может иметь два (см. строку 2) параметра */
32      game.ScoreChange += new ScoreChangeEventHandler(game_ScoreChange);
33    }
34  /* Обратите внимание, что сигнатура (см. строку 35) этого метода (см. строки 36…42)
    совпадает с  сигнатурой класса ScoreChangeEventHandler: см. строку 2*/
35    private void game_ScoreChange (int newScore, ref bool cancel)
36    {
37      if (newScore < 100) System.Console.WriteLine ("Хороший счет");
38      else {
39        cancel = true;
40        System.Console.WriteLine ("Счет не может быть таким большим!");
41          }
42    }
43  }                       // Окончание класса Referee

44  // Класс для тестирования всего примера
45  public class GameTest
46  {
47    public static void Main()
48    {
49      Game game = new Game();
50      Referee referee = new Referee(game);
51    game.Score = 70;    // см. строку 37: печать Хороший счет
52    game.Score = 110;    // см. строку 40: печать Счет не может быть таким большим!
53    }
54  }
```

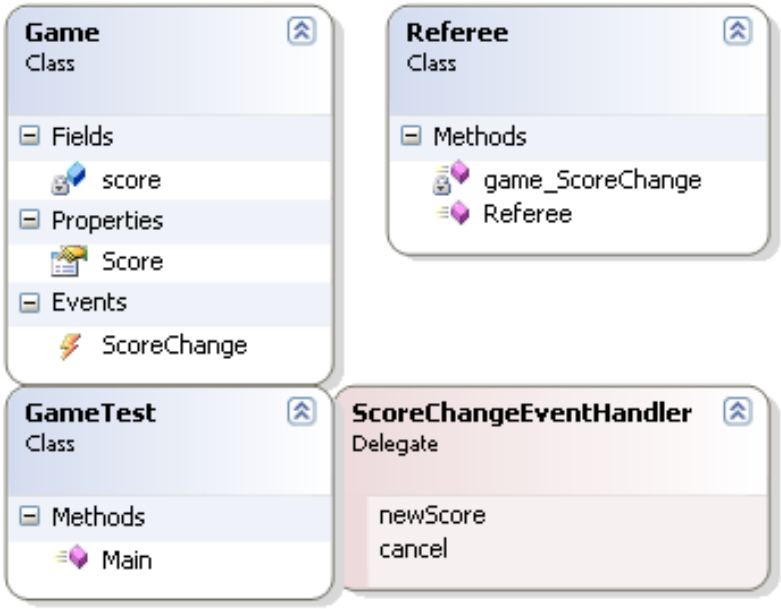

Рис. 1.1. Диаграмма классов: Листинг 1.1. Иллюстрация процесса: 1) описания, 2) генерации и 3) обработки события на языке C#

Класс **GameTest** сначала создает объект типа **Game** (**Игра**) /**строка 49**/, а затем объект типа **Referee** (**Арбитр**) /**строка 50**/, который будет следить за игрой. Изменим счет игры и посмотрим (**строки 51, 52**), как на это отреагирует **Арбитр**. В нашем примере **Игра** ничего не знает о существовании **Арбитра**, а просто позволяет любому классу следить за изменением счета и реагировать на любое изменение.

Ключевое слово **event** /**строка 6**/ скрывает все методы класса-делегата **Game** /**строки 3…25**/.за исключением «**+=**»и «**-=**» от всех классов кроме того класса, в котором это событие объявлено. Эти операторы «**+**»**=** (и «**-=**»)позволяют добавлять /**строка 32**/ (и убирать) сколько угодно обработчиков **game_ScoreChange** /**строки 34…42**/.для данного события **ScoreChange** /**строка 20**/.

В реальных задачах с необходимостью обрабатывать события можно столкнуться при разработке графического интерфейса пользователя, где роль Игры будет выполнять графический интерфейс (**GUI**), генерирующий многочисленные события (**z.B., стандартное событие Click**) в ответ на действия пользователя (**z.B., кликанье мышкой**), а роль **Арбитра** - форма, которая должна обрабатывать эти события **[3, С.740-748]**.

> Впервые делегаты появились в **Microsoft Visual J++** и стали причиной длительных технических и юридических споров между компаниями Sun и Microsoft.

См. Терминология дисциплины, ч.1, термин 20

См. Терминология дисциплины, ч.3, термин 21

См. Терминология дисциплины, ч.2, термин 56

1.6. Перечисления (Enums) [3, С.222-229]

Перечисления позволяют задать группу объектов, например**:**

- Описание: **C#**

```
public enum Direction {North, East, West, South};
```

Пример использования: **C#**

```
Direction wall = Direction.North;
```

Очень удачная конструкция, поэтому вопрос состоит не в том, почему в **C#** перечисления используются, а в том, почему в **Java** про них забыли? На языке **Java** вам придется написать:

- Описание: **Java**

```
public class Direction
 {
        public final static int NORTH = 1;
        public final static int EAST = 2;
        public final static int WEST = 3;
        public final static int SOUTH = 4;
}
```

Использование будет выглядеть так: **Java**

```
int wall = Direction.NORTH;
```

Мало того, что описание в **Java** выглядит более громоздко, так оно еще и не является безопасным, так как ничто не мешает вам случайно присвоить объекту (переменной) **wall** любое целое значение, компилятор не выдаст никакого предупреждения.

Еще одно преимущество **C#** при работе с перечислениями состоит в том, что если в режиме отладки кода вы поставите точку останова на перечислении, то отладчик выполнит трансляцию значение, так что вы увидите понятное название, а не абстрактное число. Например, если в приведенном ниже примере поставить точку останова на оператор **if**:

```
Direction direction = Direction.North | Direction.West;
  if ((direction & Direction.North) != 0)
     ....
```

то вы увидите понятное направление, а не число **5**.

Наиболее вероятное объяснения отсутствия перечислений в **Java**, видимо заключается в том, что аналогичную конструкцию можно реализовать с помощью класса. Философия **Java** гласит: если это можно реализовать через класс, то зачем вводить в язык новую конструкцию? Преимущество такого подхода состоит в упрощении языка, соответственно, на его изучение требуется меньше времени, - не нужно держать в голове множество спосособов достижения результата. Конечно, **язык Java выигрывает у C++** во многих аспектах и прежде всего упрощает написание кода за сч ет отсутствия: **a)** указателей, **b)** заголовочных файлов, **c)** множественного наследования. Однако **отсутствие** рассмотренных выше конструкций: **1) перечислений, 2) свойств и 3) событий**, наоборот усложняет кодирование **(NB)**.

См. Терминология дисциплины, ч.3, термин 20

1.7. Коллекции и оператор foreach /см. на с. 33 раздел 2.4.1/ [3, С.358-229, С.422-423]

C# предлагает *стенографически* короткую форму записи циклов с помощью оператора **foreach**. Особенно удобно использовать этот оператор при работе с коллекциями, так как он поддерживает **согласованность обращений к элементам 1) коллекций** (классы коллекций из пространства имен **System.Collection** .NET Framework) **или 2) массивов**:

На языке **Java или C++** циклы выглядят следующим образом:

```
1.  while (! collection.isEmpty())  // обращение к элементам колекции
    {
      Object o = collection.get();
      collection.next();
        ...
2. for (int i = 0; i < array.length; i++)...  // обращение к массиву
```

В **C#**:

```
1. foreach (object o in collection)...     // обращение к элементам колекции
2. foreach (int i in array)...             // обращение к массиву
```

В **C#** цикл **for** также будет работать с коллекциями (массивами). **Объект** коллекции имеет метод **GetEnumerator()**, который возвращает объект-нумератор. Этот объект имеет метод **MoveNext** и свойство **Current**.

См. Терминология дисциплины, ч.3, термин 29

1.8. Структуры (structures) /см. на с. 27 раздел 2.1/ [3, С.701-708]

Структуры (**см. Терминология дисциплины, ч.3, термин 61**) в **C#** полезно воспринимать как конструкцию, привносящую **дополнительную элегантность в систему типов языка**, нежели как просто способ получения более эффективного кода.

В **C++** память для объекта структуры **и** класса может выделяться как: **1) в стеке**, так и **2) в куче** (**heap**). В **C#** **структуры** всегда создаются **в стеке** (**см. Терминология дисциплины, ч.1, термин 67*а***) , а **классы** (**см. Терминология дисциплины, ч.2, термин 23**) - **в куче**. Конечно, использование структур позволяет получить более эффективный код.

```
public struct Vector   //  C#; ключевое слово struct определяет типы-значения [1]
{
   public float direction;
   public int magnitude;
}

Vector[ ] vectors = new Vector [1000];  //  C#
```

В приведенном примере память для **1000** векторов будет выделена одним массивом, что гораздо эффективнее, нежели объявлять класс, а затем с помощью цикла **for** создавать **1000** отдельных векторов. Эффективность достигается за счет того, что мы объявляем массив структур так, как могли бы объявить массив целых чисел в **C#** или **Java**:

```
int[ ] ints = new ints[1000];
```

C# просто расширяет рамки набора примитивных типов, имеющихся в языке. На самом деле, **C#** трактует все примитивные типы как структуры. Тип **int** является ничем иным, как псевдонимом для структуры **System.Int32**, тип **long** – псевдонимом для структуры **System.Int64** и т.д. Конечно компилятор может специальным образом обрабатывать эти и другие примитивные типы, но сам язык не делает между ними различий. В следующем **разделе 1.9** рассмотрено, какие преимущества можно извлечь в **C#** в связи с этим.

---

[1] **ключевое слово class определяет ссылочные типы**

1.9. Унификация типов [3, С.141-196]

В большинстве языков программирования есть примитивные типы (**int, long, char** и т.д.) и типы данных более высокого уровня, набираемые из примитивных типов. Однако зачастую очень удобно было бы обходиться с примитивными и сложными типами данных одинаково. **Java** обращается с примитивными типами так же, как и **C** или **C++,** но предлагает свой упакованный класс для каждого примитива**:** тип **int** упаковывается в класс **Integer**, **double** - в **Double** и т.д. В **C++** с помощью шаблонов можно написать код, который сможет работать с любыми типами, если указанные операции имеют смысл для данного типа.

В **C#** эта проблема решается по-другому. В предыдущем разделе отмечалось, что примитивные типы, например **int,** являются псевдонимами для соответствующих структур. Но так как структуры имеют методы, определенные для класса **Object**, то можно написать следующий код:

```
int i = 5;
System.Console.WriteLine (i.ToString());
```

Если необходимо обращаться со структурой как с объектом, то **C# упакует** ее в объект, а затем **распакует** ее, когда понадобится обратиться к ней, как к структуре:

```
Stack stack = new Stack();      // см. Листинг 2.3. на с. 35
stack.Push (i);                 // упаковываем int
int j = (int) stack.Pop();      // распаковываем int
```

Если не считать операции приведения типа, необходимой при распаковывании структур, имеем совершенно прозрачный способ обработки взаимосвязей между структурами и классами. Следует помнить, что упаковка не влечет за собой создания объекта, поэтому **CLR** может применить дополнительную оптимизацию по отношению к упакованному объекту.

> Разработчики **C#** наверняка думали о целесообразности использования шаблонов. Можно предположить, что **они решили** отказаться от шаблонов **по двум основным причинам.** Первая - сохранение чистоты языка, шаблоны было бы тяжело связывать с объектно-ориентированными свойствами, - они открывают перед пользователями слишком много сбивающих с толку возможностей, и их тяжело совместить с рефлексией типов. Вторая причина заключается в том, что шаблоны не будут слишком полезны сами по себе, если библиотечные классы в **.NET**, например коллекции, не будут их поддерживать. А если включить поддержку шаблонов в библиотеки **.NET**, - это значит, что **20** с лишним языков программирования, использующих библиотеки **.NET**, тоже должны поддерживать работу с шаблонами, что слишком сложно реализовать технически.
>
> Андерс Хейлсберг заявлял, что шаблоны в **C#** маячат на горизонте, но в первом релизе не появятся в связи с указанными выше сложностями. Интересно заметить, что спецификация **IL** составлена так, что **IL** способен поддерживать шаблоны.

См. Терминология дисциплины, ч.2, термин 65

См. Терминология дисциплины, ч.1, термины 65 и 75

1.10. Перегрузка операторов /см. на с. 30 раздел 2.2.3/ [3, С.563-572]

Перегрузка операторов дает возможность программистам обращаться со сложными типами так же естественно, как и с примитивными **(int, long, ...)**. **C#** подходит к перегрузке операторов строже, чем **C++**, однако здесь вполне можно реализовать перегрузку операторов для таких классических случаев, как операции с комплексными числами.

В **C# оператор ==** является **не**виртуальным (**операторы не могут быть виртуальными**) методом, который позволяет сравнивать объекты двух классов по ссылке. Когда вы описываете класс, вы можете определить свой собственный оператор **==**. Если вы собираетесь использовать класс как коллекцию, вы должны реализовать интерфейс **IComparable**. Этот интерфейс требует реализации метода **CompareTo**, который должен возвращать положительное, отрицательное число или 0, если "**this**" **больше**, **меньше** или **равно** объекту. Если вы решите сделать синтаксис сравнения более удобным, вы можете определить методы **<**, **<=**, **>=**, **>**. Примитивные числовые типы (**int, long и т.д.**) реализуют интерфейс **IComparable**.

Ниже приведены простейшие примеры реализации операторов сравнения:

```
public class Score : Icomparable // класс Score реализует интерфейс Icomparable
{
   int value;
   public Score (int score)
  {
      value = score;
   }
   public static bool operator == (Score x, Score y)
  {
      return x.value == y.value;
   }
   public static bool operator != (Score x, Score y)
  {
      return x.value != y.value;
   }
   public int CompareTo (object o)
  {
      return value - ((Score)o).value;
   }
}

Score a = new Score (5);
Score b = new Score (5);
Object c = a;
Object d = b;
// Для сравнения a и b по ссылке:
System.Console.WriteLine ((object)a == (object)b; // ложь
// Для сравнения a и b по значению:
```

```
System.Console.WriteLine (a == b);                    // истина
// Для сравнения c и d по ссылке:
System.Console.WriteLine (c == d);                    // ложь
// Для сравнения c и d по значению:
System.Console.WriteLine(((IComparable)c).CompareTo (d) == 0);  // истина
```

Вы также можете добавить в класс **Score** операторы **<, <=, >=, >.** В процессе компиляции **C#** проверит, что вы не забыли определить логически парные операторы: **!=** и **==**, **>** и **<**, **>=** и **<=**.

1.11. Полиморфизм [3, С.652-699]

Для реализации полиморфизма в объектно-ориентированных языках служат **виртуальные методы**.

Полиморфизм позволяет переопределить реакцию производного класса на метод, определенный в базовом классе.

Сигнатуры (**См. Терминология дисциплины, ч.2, термин 53**) этих методов **одинаковы**, а определения (то есть тела) методов **разные**.

В **Java** по умолчанию все методы являются виртуальными.

В **C#**, как и в **C++**, виртуальные методы должны быть обозначены ключевым словом **virtual**. Кроме того, в **C#** следует использовать ключевое слово **override**, чтобы показать, что вы переопределяете в производном классе (или определяете, для абстрактных методов) метод базового класса.

```
Class B
{
   public virtual void foo() { }
}
```

```
Class D : B
{
   public override void foo() { }
}
```

Попытка подменить **не**виртуальный метод в производном классе приведет к появлению ошибки на стадии компиляции, если вы не добавите в его описание ключевое слово **new** (**См. Терминология дисциплины, ч.3, термин 41**), показывающее, что данный метод умышленно скрывает метод базового класса.

```
Class N : D
 {
   public new void  foo() {}
 }

N n = new N();
n.foo();              // вызов N's foo
((D)n).foo();         // вызов D's foo
((B)n).foo();         // вызов D's foo
```

Обязательное использование ключевого слова **override** облегчает восприятие исходног о

кода и это безусловно плюс по сравнению с **Java** и **C++**.

Необходимость указания ключевого слова **virtual** имеет свои плюсы и минусы. С одной стороны, это несколько повышает эффективность выполнения кода (**это плюс**), а с другой, препятствует расширяемости кода (**это минус**).

См. Терминология дисциплины, ч.3, термины 46, 74

См. Терминология дисциплины, ч.1, термины 60, 85, 53

См. Терминология дисциплины, ч.2, термины 43, 7

1.12. Интерфейсы [3, С.652-699]

Интерфейсы в **C#** напоминают интерфейсы в **Java**, но обладают большей гибкостью. Класс может "**явно**" реализовать интерфейс, например:

```
public interface ITeller
{
   void Next();
}
```

```
public interface IIterator
{
   void Next();
}
```

```
public class Clark : ITeller, IIterator
{
   void ITeller.Next()
   {    ……
   }
   void IIterator.Next ()
   {    ……
   }
}
```

С этой возможностью может быть связано **два преимущества**.

Во-первых, класс может реализовать несколько интерфейсов, не опасаясь конфликта имен.

Во-вторых, можно "**спрятать**" метод, если он не представляет ценности для большинства пользователей класса.

Явно реализованные методы могут быть вызваны путем приведения интерфейса к типу класса:

```
Clark clark = new Clark();
((ITeller)clark).Next();
```

См. Терминология дисциплины, ч.3, термин 35

См. Терминология дисциплины, ч.1, термин 32

См. Терминология дисциплины, ч.2, термин 69 (сравнить с термином 22)

1.13. Управление версиями [3, С.678-681]

Вопросу управления версиями в среде **.NET** уделяется особое внимание. Большая часть соглашений на эту тему относится к сборкам. Предусмотрены даже весьма экзотичные возможности, как то выполнение различных версий одной и той же сборки в одном процессе.

Язык **C#** препятствует возникновению программных сбоев в случае появления новых версий программных модулей (в первую очередь это относится к библиотекам **.NET**). Этот вопрос подробно освещен в техническом описании **C#**, рассмотрим его на примере.

Пусть в **Java** мы используем класс **D**, производный от класса **B**, распространяемого вместе с виртуальной машиной **Java**. Класс **D** имеет метод **foo**, который на момент выпуска нашего кода в родительском классе **B** отсутствовал. Затем появилась новая версия виртуальной машины, в которой класс **B** получил метод **foo**, и мы инсталлировали эту библиотеку на компьютер, где использовался наш класс **D**. Это может привести к сбою программы, в которой использовался класс **D**, так как в новой реализации класс **B** попытается виртуально вызвать метод класса **D**, в результате будет выполнено действие, не предусмотренное в классе **B**.

В **C#** метод **foo** в классе **D** в подобной ситуации окажется описанными без модификатора **override**, поэтому **D.foo** скроет **B.foo**, а не переопределит его!

Интересная выдержка из справочного руководства по **C#**: "**C#** решает проблему контроля версий, вынуждая разработчиков четко выражать свои намерения". Разработчики могут точно выразить свою цель с помощью ключевого слова **override**, с другой стороны, компилятор может автоматически сгенерировать его, проверяя, что метод фактически переопределен. Это означает, что разработчики могут использовать стиль **Java** без ключевых слов **virtual** и **override**, все равно контроль версий будет работать корректно.

1.14. Модификаторы параметров [3, С.468,сл.]

A. Модификатор "ref"

C# (в отличие от **Java**) позволяет передавать параметры по ссылке. Проиллюстрировать такой способ передачи параметров проще всего на следующем примере. В отличие от **C++** модификатор следует использовать не только в описании метода, но и при его вызове:

```
public class Test
{
   public static void Main()
   {
      int a = 1;
```

```
        int b = 2;
        swap(ref a, ref b);                          // вызов метода
    }

    public static void swap (ref int a, ref int b) );   // описание  метода
    {
        int temp = a;
        a = b;
        b = temp;
    }
}
```

См. Терминология дисциплины, ч.3, термин 52

B. Модификатор "out"

Ключевое слово **out** является естественным дополнением к модификатору **ref**.

Если модификатор **ref** требует, чтобы параметру было присвоено значение до того, как он будет передан методу, то модификатор **out** заставляет компилятор проверять, что параметру будет присвоено значение, прежде чем метод вернет управление.

См. Терминология дисциплины, ч.3, термин 45

C. Модификатор "params" /см. на с. 34 раздел 2.4.3/

Модификатор **params** может быть задан для последнего параметра метода, он показывает, что методу может быть передано *произвольное* количество параметров соответствующего типа. Например:

```
public class Test
{
    public static void Main()
    {
        Console.WriteLine (add(1, 2, 3, 4).ToString());
    }

    public static int add(params int[ ] array)
    {
        int sum = 0;
        foreach(int i in array) sum += i;   //см. на с. 14 раздел 1.7
        return sum;
    }
}
```

См. Терминология дисциплины, ч.3, термин 47

Один из наиболее неожиданных сюрпризов, с которым вы непременно столкнетесь при изучении **Java**, состоит в отсутствии **возможности** передачи параметров по ссылке. Однако, после некоторого начального замешательства, вы обнаруживаете,

что вполне можете обойтись без этой возможности.

Почему же создатели **C#** решили реализовать эту **возможность?**

Java значительно упрощает жизнь разработчиков, устанавливая четкие ограничения на способы передачи параметров.

Это особенно заметно при сравнении с **C++**, где параметры можно передавать: **1)** и по значению, **2)** и по ссылке, а учитывая наличие указателей, **код программы может стать чрезвычайно запутанным**.

C# допускает передачу параметров по ссылке, но заставляет разработчиков явно указывать на использование ссылок как в описании метода, так и при его вызове, тем самым упрощая чтение кода, т.е. достигается тот же эффект, что и **Java**, но с большей выразительностью.

В **C#** стараются не заставлять программистов искать обходные пути, когда им нужно решить конкретную задачу.

1.15. Атрибуты /см. на с. 30 раздел 2.2.4/ [3, С.757-762]

В **C#**, как и в **Java**, можно получить информацию **об уровне доступа** к любому полю в откомпилированном коде.

В **C#** эта возможность была расширена и получила общий характер, теперь вы можете скомпилировать произвольную прикладную информацию, связав ее с любым фрагментом кода**:** классом, методом, полем и даже с отдельным параметром. Доступ к этой информации можно получить во время исполнения программы. Ниже приведен упрощенный пример класса, иллюстрирующий использование атрибутов **(см. Терминология дисциплины, ч.2, термин 4)**:

```
[AuthorAttribute("Ivan Petrov")]
class A
{
   [Localizable(true)]
   public String Text
   {
      get
         {
return text;
         }
      ...
   }
}
```

Для включения дополнительной информации о классах и методах **Java** использует комбинацию комментариев /** */ и **@tag**, но эта информация (за исключением **@deprecated**) не попадает в откомпилированный код.

Все атрибуты в **C#** происходят от абстрактного класса **System.Attribute**. **C#** использует

встроенный атрибут **ObsoleteAttribute**, который заставляет компилятор генерировать предупреждения в случае использования устаревшего кода, и атрибут **ConditionalAttribute**, управляющий условной компиляцией. Новые **XML**-библиотеки **Microsoft** используют атрибуты для определения правил сериализации полей в формат **XML**. Это означает, что вы можете легко выгрузить класс в формат **XML (Extensible Markup Language – язык XML –расширяемая спецификация языка, предназначенного для создания веб-страниц)**, а затем снова восстановить его.

Другое предназначение атрибутов заключается в возможности создания по-настоящему мощных программ для просмотра классов. Подробнее вопросы создания и использования атрибутов рассмотрены в Справочном руководстве по языку **C#**.

Атрибут позволяет добавить декларативную информацию **C#** к элементам кода: сборкам, классам, структурам, элементам классов и структур, формальным параметрам и возвращаемым значениям. Декларативная информация делает код более удобным **[3, С.757-762].**

1.16. Операторы выбора /см. на с. 33 раздел 2.4.2/ [3, С.292-303]

Для управления оператором **switch** в **C#** можно использовать: **1)** целочисленные типы, **2)** символы, **3)** перечисления (**enum**) и **3)** строки (в отличие от **C++** и **Java**).

Если в **C++** вы пропустите **break** после блока оператора **case**, то будет выполнен следующий блок **case**. Непонятно почему в **C++** и **Java** по умолчанию действует такая логика, но в **C#** это к счастью не так.

См. Терминология дисциплины, ч.3, термин 5, 7, 62

1.17. Встроенные типы [3, С.142-196]

В **C#** набор примитивных типов соответствует набору типов **Java**, но к этому набору были добавлены "**беззнаковые**" типы. Таким образом, в **C#** определены: **sbyte, byte, short, ushort, int, uint, long, ulong, char, float** и **double**.

Единственным сюрпризом в **C#** оказалось появление **12-байтовых** чисел с плавающей точкой (**decimal**), которые могут использовать преимущества последних моделей процессоров.

См. Терминология дисциплины, ч.3, термин 14 и др.

См. Терминология дисциплины, ч.1, термин 75

См. Терминология дисциплины, ч.2, термин 65

1.18. Модификаторы полей [3, С.188-191]

Снова можно отметить, что модификаторы полей в **C#** в основном повторяют соответствующие элементы в **Java**.

Для выделения полей, которые нельзя модифицировать, **C#** предлагает использовать

модификаторы **const** и **readonly**.

Модификатор **const** действует аналогично модификатору **final** в **Java**, и при компиляции соответствующее значение становится частью **IL**-кода.

В случае использования модификатора **readonly** значение поля вычисляется на стадии выполнения программы. В этом случае вы можете установить новые версии библиотек, например стандартных библиотек **C#**, без перекомпиляции основного кода программы.

См. Терминология дисциплины, ч.3, термин 12, 51

См. Терминология дисциплины, ч.2, термин 42

1.19. Операторы перехода [3, С.290-292]

Здесь тоже не слишком много сюрпризов, если не считать *печально* известного **goto**. Однако в своей эволюции он ушел далеко от "зловредного" оператора **goto**, знакомого нам по версиям **Basic** 20-летней давности.

Оператор **goto** должен указывать: **1)** на метку **2)** или одну из опций в операторе **switch**.

Вариант указания на метку аналогичен случаю использования оператора **continue:label** в **Java**, но допускает чуть больше свободы. Так оператор **goto** может указывать на любую точку кода в пределах области его видимости, т.е. в рамках метода или блока **finally**, если он определен в нем. Он не может перепрыгнуть в оператор цикла или выйти за пределы блока **try**, пока не будет выполнен соответствующий блок **finally**. Оператор **continue** в **C#** аналогичен **continue** в **Java**, но **не может** указывать на метку.

См. Терминология дисциплины, ч.3, термин 30, 62, 66, 25, 13

1.20. Сборки, пространства имен и уровни доступа [3, С.50-54, 589-612]

В **C#** вы можете собирать компоненты исходного кода (классы, структуры, делегаты, перечисления и т.д.) в файлы, пространства имен и сборки.

Пространства имен - это не что иное, как способ упрощения синтаксиса при работе с длинными именами классов. Например, вместо того, чтобы обращаться к классу **Genamics.WinForms.Grid**, вы можете описать класс как **Grid** и поместить его в пространство имен:

```
namespace Genamics.WinForms
{
   public class Grid
   {
      ....
   }
}
```

В классы, использующие **Grid**, вы можете импортировать соответствующее пространство имен с помощью ключевого слова **using** (**См. Терминология дисциплины, ч.3, тер-**

мин 73), тогда можно избежать ссылок с указанием полного имени класса.

Сборки - это .EXE и .DLL файлы, которые генерируются после компиляции соответствующих проектов (файлов). Среда исполнения **.NET** использует конфигурируемые атрибуты и правила контроля версий, встроенные в сборки, что в значительной степени упрощает распространение приложений, - не нужно больше копаться в реестре, - достаточно скопировать сборку в нужный каталог, и можно запускать приложение. Сборки определяют также границы действия типов, что препятствует появлению конфликта типов и позволяет запускать несколько версий сборок в одном процессе. Каждый файл может включать множество классов и множество пространств имен. С другой стороны, пространство имен может охватывать несколько файлов. Так что в этом отношении **C#** предоставляет большую свободу.

В **C#** определено пять уровней доступа: **private, internal, protected, internal protected** и **public**. **Private** и **public** имеют тоже значение, что и в **Java**, но нужно помнить, что в **C#** по умолчанию действует уровень доступа **private**. Уровень доступа **internal** соответствует рамкам сборки, а не пространства имен, что характерно для **Java**. Уровень доступа **internal protected** соответствует **protected** доступу в **Java**, а **protected** доступ в **C#** соответствует уровню **private protected** в **Java**, который там принято считать устаревшей формой.

См. Терминология дисциплины, ч.1, термин 4

См. Терминология дисциплины, ч.3, термин 40

1.21. Арифметика с указателями /см. на с. 32 раздел 2.3/

Операции с указателями в **C#** разрешены только в методах, отмеченных модификатором **unsafe**. Если указатели указывают на объекты, которые могут быть удалены сборщиком мусора, компилятор принудительно помечает их с помощью слова **fixed**, чтобы "пришпилить" такие объекты. Сборщик мусора (**garbage collector – см. Терминология дисциплины, ч. 1, термин 25**) периодически анализирует ссылки на объекты, чтобы иметь возможность удалять ненужные объекты и освобождать память в **куче**, но если это произойдет, когда вы работаете с обычными указателями, возникнет проблема. Слово **unsafe** ("**небезопасный**") очень точно отражает смысл соответствующих участков кода, заставляя разработчиков внимательно относиться к работе с указателями.

См. Терминология дисциплины, ч. 3, термин 71, 26

1.22. Некоторые синтаксические отличия языков C# и Java

Таблица 1.1

Наименование (смысл) синтаксической конструкции	**C#**	**Java**	**# раздела**
Базовый и **производный** классы	Class **Student** : **Person**	Class **Student extends Person**	

Реализация **интерфейса**	Class Student **:** Person, **IHTMLSourse**	Class Student **extends** Person **implements HTMLSourse**	**1.12**
Обращение к методу базового класса из производного класса	**base.**Displey()	**super.**Displey()	
Объявление констант	**const**	**final**	**1.18**
Имя метода	**Main()**	**main()**	
Описание логического типа	**bool**	**boolean**	**1.9, 1.17**
Обращение к конструктору базового класса **A** из производного класса **B**	**A(int x, int y):base(int x);** {this.y = y}	**A(int x, int y)** {**super(int x);** this.y = y}	
Печать	**S**ystem**.Console.WriteLine**(…)	**s**ystem**.out.println**(…)	
Переопределение метода базового класса в производном классе (форма полиморфизма)	Переопределяе**мый** метод базового класса **явно** объявляется виртуальным (**virtual**); Переопределя**ющий** метод производного класса в заголовке содержит ключевое слово **override**	Все методы **по умолчанию** являются виртуальными	**1.11**
Доступ к закрытым переменным (полям)	Посредством специальной конструкции, называемой **Свойство**. **Свойство** содержит блоки **set** и/или **get.** Также можно воспользоваться методом-**аксессором** и методом-**мутатором**	Только посредством метода-**аксессора** и метода-**мутатора**.	**1.2**
Доступ к элементам перечислений	Посредством специальной конструкции, называемой **Индексатор**, для ее создания используется сочетание **this[ ]**	–	**1.3**
Безопасный объектно-ориентированный указатель на функцию	Используется специальное понятие **делегат** (**delegate**)	Необходимо определять внутренние адаптеры или дополнительный код для вызова нескольких методов	**1.4**
Поддержка **Событий** (**Events**)	Обеспечена **непосредственная** поддержка Событий	Для поддержки Событий используются **внутренние адаптеры**	**1.5**
Поддержка **перечислений** (**enums**)	Осуществляется посредством специальной синтаксической конструкции **enum**	–	**1.6**
Форма цикла для работы с коллекциями и массивами	Предусмотрена специальная стенографически короткая форма записи циклов с помощью оператора **foreach**	Циклы **for** и **while**	**1.7**
Встроенный тип **12-байт-х** чисел с плавающей точкой	Предусмотрено ключевое слово **decimal**	–	**1.17**
etc., etc.			

2. Язык C# и программирование на платформе .NET

2.1. Система типов языка C# /см. на с. 15 раздел 1.8 и на с. 16 раздел 1.9/

2.1.1. Ссылочные типы и типы-значения

Система типов среды **CLR** (**C**ommon **L**anguage **R**untime) состоит из **двух** основных частей**:**

- **ссылочный тип** — это тип, память для которого выделяется **в куче** и используется для большинства объектов.
- **тип-значение** — это тип, память для которого выделяется **в стеке** или другом объекте. Тип-значение обычно используется для предопределенных числовых типов и других традиционных данных.

Система типов **двух** разновидностей представляет проблему для таких языков программирования, как **C#**, который стремится предоставить **единую** систему типов там, где это только возможно. Если ссылочные типы и типы-значения были бы не совместимы и существовали раздельно, то невозможно было бы создать приведенный ниже код.

```
int value = 55;
double precision = 0.35;
Console.WriteLine("Result: {0}, {1}", value, precision);
```

Для использования этого кода требуется выполнить перегрузку метода **WriteLine** для типов **string**, **int** и **double**.

Среда выполнения предлагает способ преобразования типа-значения в ссылочный тип, который называется упаковкой (**boxing – См. Терминология дисциплины, ч.1, термин 7**). **Упаковка** — это оболочка ссылочного типа для типа-значения, которая позволяет передавать тип-значение как любой другой ссылочный тип в виде параметра типа **object** (**См. Терминология дисциплины, ч.3, термин 43**).

Там, где это возможно, язык **C#** автоматически и совершенно незаметно для пользователя выполняет упаковку, и поэтому ее не следует указывать явно. При использовании типа-значения в том месте, где требуется упаковка, компилятор автоматически генерирует код упаковки. (Это делается за счет присвоения типу-значению объектного типа или приведения к интерфейсному типу.) Это позволяет пользователю создавать код без учета типа аргументов, независимо от того, являются ли они ссылочными типами или типами-значениями.

```
Console.WriteLine("Result: {0}, {1}", value, precision);
```

После упаковки типа-значения его экземпляр становится непроницаемым для **C#**, т.е. не существует разницы между упакованным целочисленным значением, упакованным значением с плавающей запятой и определенным пользователем упакованным значением. Это упрощает пользовательскую модель и означает, что нет никакого способа изменения значения упакованного экземпляра, кроме распаковки его (**unboxing – См. Терминология дисциплины, ч.1, термин 80**) исходного типа, изменения значения исходного типа и последующей упаковки. Эти операции снижают производительность при сохранении значений упакованных типов в классах-коллекциях.

2.1.2. Определенные пользователями типы

В языке **C# ссылочные типы** создаются с помощью ключевого слова **class**,
а **типы-значения** — с помощью ключевого слова **struct**.
Хотя типы-значения неявно наследуются от типа **Object** (**см. Терминология дисциплины,**

ч.1, термин 52), это делается только посредством упаковки.

Типы-значения в основном используются для представления: **1)** комплексных значений, **2)** очень больших значений (BigNums) или **3)** векторов.

Кроме того, **C#** позволяет определять интерфейсы с помощью ключевого слова **interface**, а также перечисляемые константы и битовые флаги с помощью ключевого слова **enum**.

2.2. Компонентное программирование [3, С.32-35, С.50-54]

Язык **C#** предназначен для создания компонентов, поэтому он имеет много средств, упрощающих создание и использование компонентов.

2.2.1. Свойства /см. на с. 8 раздел 1.2/

Пользователям нравится простота использования объектов. Например, при работе с кнопкой формы пользователи могут легко создать код надписи на кнопке с помощью прямого доступа к полю надписи.

```
cancelButton.caption = "Cancel";
```

Это очень эффективный код, потому что он непосредственно изменяет значение поля.

К сожалению, эта простая модель усложняет действия разработчика программы, потому что теперь нужно активизировать измененное состояние надписи. Для каждого изменения значения поля разработчику платформы нужно предусмотреть еще один этап активизации изменения.

```
cancelButton.caption = "Cancel";
cancelButton.Paint();
```

Это не очень оптимальное решение, потому что разработчику программы нужно помнить о необходимости вызова метода **Paint()** для внесения такого изменения.

Другим способом активизации изменения является использование шаблона проектирования свойства. Вместо предоставления программисту поля с надписью разработчик платформы может предложить функцию **SetCaption()**, которая автоматически выполнит операцию **Paint()**.

```
string GetCaption();
void SetCaption(string caption);
```

В таком случае программисту достаточно будет создать следующую строку кода:

```
cancelButton.SetCaption("Cancel");
```

Это прекрасный способ, но теперь программисту нужно выполнить гораздо больше действий, чем требует простая организация доступа к значению поля. Ему теперь нужно найти функции **Get** и **Set** класса **Button**, догадаться об их связи и создать более сложный код.

Свойства в языке **C#** позволяют разработчику платформы использовать шаблон проектирования и предоставить программисту более простую модель поля. Теперь **автор класса-кнопки** может использовать приведенный ниже код.

```
public  string Caption
 {
        get
        {
                return caption;
        }
        set
        {
                caption = value;
                Paint();
```

```
    }
}
```

А **пользователь этой кнопки** для изменения надписи на ней может использовать следующий код:

```
cancelButton.Caption = "Cancel";
```

Итак, свойства значительно упрощают работу с компонентами.

2.2.1.1. Особенности реализации

Среда выполнения **CLR** позволяет сделать выбор способа применения свойств достаточно гибким. В языке **C#** они представлены как "виртуальные поля". Хотя автор свойства создает код для получения и указания значения свойства, а компилятор представляет их в виде **get-** и **set-**методов, с точки зрения пользователя, свойство ведет себя как поле.

В результате такие ключевые слова, как **public, virtual** и **abstract**, могут применяться только на уровне свойства, а не на уровне методов доступа. Это усложняет некоторые полезные идиомы (например, наиболее популярную идиому "public get-метод и protected set-метод"), но упрощает пользовательскую модель.

2.2.2. Индексаторы /см. на с. 9 раздел 1.3/

На платформе **.NET Framework** существует класс **ArrayList**, который представляет собой массив с динамически изменяемым размером. Для доступа к значениям массива **ArrayList** предусмотрены следующие функции:

a) object **GetItem**(int index);

b) void **SetItem**(int index, object value).

Как и в предыдущем разделе о свойствах, эти функции усложняют код, но, что гораздо важнее, позволяют работать с классом **ArrayList** иначе, чем с массивом. Для использования модели массива в других классах в языке **C#** предусмотрено создание **индексатора**, который позволяет пользователю непосредственно индексировать массив **ArrayList**.

```
public object this[int index]
{
        get
        {
        return items[index];
        }
        set
        }
        items[index]  = value;
        }
}
```

2.2.2.1. Особенности реализации

В среде выполнения поддерживаются параметрические свойства, т.е. свойства, которые принимают параметры. В языке Visual Basic .NET класс может иметь несколько параметрических свойств, и в этом нет ничего особенного.

В языке **C#** параметрическое свойство выражается в виде виртуального массива, который позволяет индексировать объект как массив. Например, класс-коллекция состоит из элементов, поэтому индексирование является прекрасным способом организации доступа к ним.

В языке **C#** поддерживается только одно параметрическое свойство, хотя его можно перегрузить параметрами разного типа.

2.2.3. Перегрузка операторов /см. на с. 17 раздел 1.10/

В языке **C#** поддерживается перегрузка операторов, что позволяет пользователям создавать новые типы, которые будут действовать аналогично имеющимся типам, непосредственно поддерживаемым средой выполнения. (Интересно, что тип **decimal** не поддерживается непосредственно механизмом выполнения, а считается создаваемым пользователем типом). Пользователи могут перегружать: **1)** унарные и **2)** бинарные операторы, **3)** а также операторы сравнения. Кроме того, они могут перегружать **4)** значения **true** и **false** для типов с неопределенными значениями.

2.2.3.1. Особенности реализации

Перегрузка операторов значительно усложняет язык программирования, но она имеет огромное значение для создания новых типов, которые могут на равных использоваться вместе со встроенными типами. Вероятно, самым лучшим примером использования механизма перегрузки являются типы данных **SQL** (**S**tructured **Q**uery **L**anguage; язык структурированных запросов, язык **SQL**), которые работают аналогично встроенным типам, но также поддерживают **SQL**-концепцию обработки неопределенных значений (**nullability**) для логических выражений или операций сравнения. (Иначе говоря, при сравнении двух экземпляров объекта **SQLint**, которые равны **null**, они не считаются равными, потому что по правилам языка **SQL null != null**.)

2.2.4. Атрибуты /см. на с. 22 раздел 1.15/

Атрибуты позволяют аннотировать (сопровождать) код информацией, которая сохраняется в метаданных и доступна во время выполнения.

В языке программирования **C#** атрибуты указываются **в квадратных скобках**.

[MarshalAs(UnmanagedType.LPTStr)]

Если атрибут не имеет параметров, то **круглые** скобки можно опустить.

[NonSerialized]

Если элемент имеет несколько атрибутов, то их можно записать в виде набора отдельных атрибутов, каждый из которых заключен в квадратные скобки, или в виде списка элементов, разделенных запятой и заключенных в квадратные скобки.

Для определения имени атрибута компилятор **C#** сначала ищет производный класс от класса **System.Attribute** с указанным точным именем. Если такой класс не найден, то компилятор добавит к указанному имени слово **Attribute** и снова попытается найти класс с таким именем.

В некоторых случаях элемент атрибута имеет двусмысленное имя, как в показанном ниже примере.

[MyAttribute("Value")] public int **MyMethod();**

Здесь атрибут может относиться либо к методу, либо к возвращаемому значению. Для исключения этой двусмысленности в языке **C#** во всех двусмысленных ситуациях используется элемент по умолчанию и пользователю разрешается указать другой элемент в случае необходимости.

[returnValue: MyAttribute("Value")] public int **MyMethod();**

2.2.5. Делегаты /см. на с. 10 раздел 1.4/

Делегаты применяются для инкапсуляции вызываемой функции и методов экземпляра вместе с самим экземпляром. В языке программирования **C#** все делегаты являются широковещатель-

ными (**multicast delegates**), т.е. могут относиться ко всем перегруженным версиям метода.

2.2.5.1. Особенности реализации

В языке **C#** для упрощения операций с делегатами используется специальный синтаксис. Для внесения нового делегата в список делегатов используется оператор «**+=**», а для удаления – оператор – «**-=**».

Допустим, пользователь пишет:

```
myDelegate  +=  new MyEventHandler(myRoutine);
```

Компилятор превращает эти строки в следующий код:

```
myDelegate = (MyEventHandler) Delegate.Combine(myDelegate, new MyEventHandler(myRoutine));
```

Аналогично, удаление компилятор может выполнять с помощью оператора «**-=**» или метода **Delegate.Remove()**.

2.2.6. События /см. на с. 11 раздел 1.5/

Событие можно представить себе как свойство над делегатом, которое предоставляет возможность управления способом доступа и ограничения доступа к переменной. В языке программирования **C#** предусмотрено два способа обработки событий. Объявление события выглядит очень просто:

```
public  event  MyEventHandler  OnClick;
```

В этой строке кода объявляется **закрытый** делегат **OnClick** и создается событие **OnClick** с соответствующими функциями **add_OnClick()** и **remove_OnClick()**. Пользователь класса может создавать и удалять события с помощью тех же операторов «**+=**» и – «**-=**», которые использовались вместе с делегатами, но теперь на основе методов **add_OnClick()** и **remove_OnClick()**.

Этот простой синтаксис прекрасно подходит для большинства случаев, но не предлагает никаких более серьезных средств контроля. Автоматически выделяемое для делегата поле (**auto-allocated delegate field)** неприменимо для объектов, которые поддерживают множество событий (с ними обычно связано несколько делегатов). (Примерами являются элементы управления **Windows Forms**) .

Поэтому в языке программирования **C#** предлагается усовершенствованный синтаксис, в котором пользователь создает функции добавления и удаления, аналогично синтаксису свойств.

```
public  event  MyEventHandler  OnClick
{
        add
        {
                // код создания
         }
        remove
        {
                // код удаления
        }
}
```

В этом случае пользователь отвечает за выделение пространства для сохранения делегата, написание кода создания и удаления, а также обеспечение безопасности при работе с потоками.

2.3. *Не*безопасный код /см. на с. 25 раздел 1.21/

В языке программирования C# пользователь может прибегнуть к ограниченному варианту "встраиваемого C" (inline C) для создания кода с использованием указателей.

Для безопасного использования указателей в среде со сборкой мусора нужно иметь механизм защиты от перемещения этих объектов сборщиком мусора. Среда выполнения позволяет закреплять объекты, которые в коде C# объявлены с помощью ключевого слова **fixed**. В **листинге 2.1** приводится пример *не*безопасного кода для суммирования байтов в массиве байтов.

Листинг 2.1. Пример использования *не*безопасного кода

```
static unsafe int DoSum(byte[ ] byteBuffer)
{
int length = byteBuffer.Length; int sum = 0;
        fixed (byte* pBuffer = byteBuffer)
        {
        byte* pCurrent = pBuffer;
                while (pCurrent < pBuffer + length)
                {
                        sum += *pCurrent;
                         pCurrent++;
                }
        }
return   sum;
 }
```

Ключевое слово **fixed** закрепляет в памяти расположение экземпляра **byteBuffer**, чтобы его всегда можно было найти по этому адресу, а код внутри блока **fixed** использует традиционную арифметику указателей. В конце блока **fixed** объект **byteBuffer** открепляется, а конструкция **fixed** содержит блок **try-finally** для гарантированного открепления.

Использование указателей в *не*безопасном блоке предотвращает проверку безопасности этого кода с точки зрения верификатора языка **IL**, а политика безопасности компьютера предотвратит выполнение этого кода в таких сценариях с загрузкой кода из сети. В текущих версиях языка программирования C# эта проблема решается на уровне сборки, поэтому использование **не**безопасного кода внутри сборки делает всю эту сборку *не*безопасной.

*Не*безопасный код обычно используется в трех случаях:

- **A.** для более эффективного взаимодействия компонентов,
- **B.** обработки существующих структур,
- **C.** а также максимального повышения производительности.

A. Более эффективное взаимодействие

Некоторые машинные и **COM**-интерфейсы используют указатели в своих определениях в таком виде, который не поддерживается при передаче данных. В этих случаях возможность использования указателей для организации взаимодействия со службами **COM Interop** (**C**omponent **O**bject **M**odel **Interop**erability – **см. Терминология дисциплины, ч.1, термин 13**) позволяет применить этот код непосредственно в коде C# без создания оболочки с помощью управляемых расширений C++ (**Managed Extensions to C++**). (Иногда такая оболочка все-таки нужна, а использовать управляемые расширения может оказаться проще, чем эквивалентный небезопасный код на языке C#, особенно в очень сложных ситуациях.)

B. Обработка существующих структур

Иногда используемые файлы могут иметь специфическую структуру, а сетевые протоколы могут задавать для сообщения специфическую структуру блока. Использование небезопасного кода позволяет расположить структуру поверх буфера чтения с диска или из сети вместо кодирования процесса чтения отдельных полей.

C. Максимальное повышение производительности

В некоторых случаях накладные расходы, связанные с доступом к данным в управляемой среде, могут быть очень высоки либо существующая библиотека может быть предназначена для непосредственного доступа к данным с помощью арифметики указателей. В таких случаях важно внимательно изучить необходимость применения такого сценария, потому что использование указателей не всегда самый лучший (или быстрый) вариант.

2.4. Другие особенности

2.4.1. Оператор foreach /см. на с. 14 раздел 1.7/

Аналогично языкам **Visual Basic .NET** и **Perl**, в **C#** предусмотрено ключевое слово **foreach**, которое применяется для перечисления объектов, реализующих интерфейс **IEnumerable.**

2.4.1.1. Особенности реализации. Часть 1

При создании контейнеров для сохранения типов значений использование интерфейса **IEnumerable** имеет нежелательный побочный эффект. Если нумератор **IEnumerable.GetEnumerator** возвращает **IEnumerator**, а свойство **IEnumerator.Current** имеет тип **Object**, то единственный способ возвращения значений нумератора основан на использовании объектов типа **Object**. Для типов значений это говорит о том, что свойство **Current** упаковывает тип значения и возвращает его клиенту, который распаковывает его. Такая упаковка и распаковка сопровождается значительным падением производительности.

Для решения этой проблемы в **C#** поддерживается альтернативный способ создания нумераторов. Если класс с помощью метода **GetEnumerator** возвращает тип, который точно соответствует **IEnumerator** (не принимая во внимание тип свойства **Current**), то компилятор для перечисления использует именно этот класс.

Для взаимодействия с другими языками в контейнерах, в которых используется этот подход, предлагается строго типизированная версия для использования внутри **C#** и закрыто реализован интерфейс **IEnumerator** для взаимодействия с другими языками среды **.NET**.

2.4.1.2. Особенности реализации. Часть 2

Оператор **foreach** содержит "**неявно явное**" ("**implicit explicit**") преобразование. Например, если пользователь использует его в виде **foreach**(**string s in listItems**), то произойдет явное преобразование типа **IEnumerator.Current** в тип, объявленный в операторе **foreach**. Поскольку конечный тип должен быть явно указан оператором **foreach** и никакого другого приведения типов не происходит, такое упрощение на основе автоматического преобразования типов вполне оправданно.

2.4.2. Оператор switch для объектов типа String /см. на с. 23 раздел 1.16/

В языке программирования **C#** пользователю разрешается создавать конструкции на основе оператора **switch** для объектов типа **String**, как показано в **листинге 2.2.**

Листинг 2.2. Пример использования оператора switch

```
switch (command)
{
```

```
case "run":
            RunAway();
break;
case "stop":
            StopDoingThat();
break;
case default:
            NotMyFault();
break;
}
```

В некоторых случаях компилятор позволяет повысить эффективность программы благодаря интернированию строк (**string interning**) во время выполнения. (Интернирование строк — это метод сохранения набора постоянных и уникальных строк во время выполнения.) Компилятор вызывает метод **string.IsInterned()** с переменной **command** и возвращает ссылку на интернированную строку, если таковая имеется, или **null** в противном случае. Если возвращается **null**, то указанная строка не упоминается ни с одним ключевым словом **case**, потому что все такие постоянные строки интернируются. Если возвращаемое значение не равно **null**, то для поиска соответствия возвращаемый экземпляр строки сравнивается со всеми строками, которые упоминаются вместе с ключевым словом **case**.

При использовании большого количества ключевых слов **case** компилятор применяет **хэш-таблицу** [**3,** С.**671**].

2.4.3. Массив params **/см. на с. 21 раздел 1.14.C/**

При создании библиотеки часто возникает проблема определения методов, которые могу принимать разное количество параметров. Простейший **способ решения этой проблемы** заключается в **перегрузке метода**.

```
int  Process(object param1);
int  Process(object param1,   object param2);
int   Process(object param1,   object param2,   object param3);
```

К сожалению, **этот подход не очень хорошо работает**, если, например, нужно создать метод с **12** параметрами. Другим вариантом решения этой задачи является создание перегруженной версии метода на основе **массива** параметров.

```
int  Process(object[ ]   values);
```

Это позволяет пользователю передавать любое количество параметров, но он должен теперь использовать их следующим образом:

```
Process(new object[ ]   {133,   13333,   "Hi",   "There",   "Mom"});
```

Этот код нельзя назвать изящным, к тому же он значительно отличается от других способов применения метода **Process()**, что усложняет его использование.

Например, программист на **C#** декорирует определение функции с помощью ключевого слова **params**:

```
int  Process(params  object[ ]   values);
```

В этом случае компилятор**: 1)** преобразует вызов в форму

```
Process(133, 13333, "Hi", "There", "Mom");
```

и **2)** автоматически создаст временный массив объектов для данного пользователя.

Такое преобразование требует дополнительных накладных расходов, **поэтому типичный спо-**

соб использования — **1)** это создание специализированных перегруженных версий для наиболее распространенных случаев (часто *максимум для трех* параметров) и **2)** применение массива параметров для обработки любых вызовов *более чем с тремя* параметрами.

2.4.4. XML-комментарии (Extensible Markup Language) [3, С.750-757]

Кроме принятого в **C++** обычного стиля комментариев, в языке программирования **C#** пользователю предоставляется возможность создавать комментарии на языке **XML** (**язык XML – расширяемая спецификация языка, предназначенного для создания веб-страниц**). В начале строк таких комментариев ставится три косые черты.

```
/// <summary>
/// Возведение в квадрат.
/// </summary>
/// <return>Возведенное в квадрат число.</return>
public double Square(double number)
```

При компиляции кода с **XML**-комментариями нужно использовать ключ компиляции **/DOC** (см. **У-МКомплекс, с. 2,** Параметры компилятора (**csc.exe**) Visual C# 2005, **пункт 7**). В таком случае компилятор гарантирует, что **XML**-комментарии содержат корректный код **XML**, проверяет корректность упомянутых имен параметров, а затем записывает **XML**-комментарии в выходной файл. Для каждого созданного **XML**-блока компилятор генерирует глобальное уникальное имя для того элемента, с которым связан этот **XML**-блок. Затем программа последующей обработки может отобразить ее в сборке, генерировать информацию о классах, а также интегрировать **XML**-комментарии для генерации окончательной документации. Кроме того, интегрированная среда разработки **Visual Studio.NET** представляет информацию из **XML**-комментариев с помощью инструмента **IntelliSense**, если полученный **XML**-файл имеет то же имя, что и сборка, и находится в одном каталоге с ней.

2.5. Пример компонента-стека

В **листинге 2.3** приводится пример компонента-стека на языке программирования **C#**. Он демонстрирует использование таких элементов, как: **1)** свойства, **2)** индексаторы и **3)** **XML**-комментарии **[4]**. Более полный пример можно создать на основе интерфейсов **ICollection и IList**.

Листинг 2.3. Пример компонента-стека

```
1  using System;
2
3  namespace ConsAppl_dotNETc266
4  {
5                  // Класс-стек общего назначения, небезопасный при работе с потоками.
6      class Stack
7      {
8          const int sizeIncrement = 10;
9          private int count = 0;
10         private object[ ] values = new object[sizeIncrement];
11
12         public virtual int Count                    // Количество элементов в стеке.
13         {
```

```
14          get
15          {
16              return count;
17          }
18      }
19
20      // Удаление верхнего элемента из стека и возвращение в стек: Верхний элемент
21      public virtual object Pop()
22      {
23          // Стек пуст,
24          if (count == 0)
25              throw new InvalidOperationException();
26          else
27          {
28              // Вернуть верхний элемент стека,
29              count--;
30              object ans = values[count];
31              values[count] = null;
32              return ans;
33          }
34      }
35
36                                      // Поместить элемент в стек: Добавляемый объект
37      public virtual void Push(object o)
38      {
39          // Расширить стек в случае необходимости
40          if (count == values.Length)
41          {
42              int newSize = values.Length + sizeIncrement;
43              object[] newValues = new object[newSize];
44              for (int i = 0; i < values.Length; i++)
45                  newValues[i] = values[i];
46              values = newValues;
47          }
48          values[count++] = o;
49      }
50
51                                      //Индексатор текущего элемента  стека.
52      public object this[int index]
53      {
54          get
55          {
56              return (values[count - index - 1]);
```

```
57          }
58      }
59
60                                  // Строковое представление стека: Строковое представление.
61      public override string ToString()
62      {
63                                                  // Преобразует все элементы стека в строки...
64          string[] args = new string[count];
65          int index = 0;
66          for (int i = count - 1; i >= 0; i--)
67          {
68              args[index] = values[i].ToString();
69              index++;
70          }
71                                                                  // ...и объединяет их.
72          return String.Join(", ", args);
73      }
74  }
75
76  class Test
77  {
78      public static void Main()
79      {
80          Stack s = new Stack();
81          for (int i = 1; i <= 15; i++)
82          {
83              s.Push(i);                              // push - проталкивать (запись в стек)
84          }
85          Console.WriteLine("\nStack s = {0}", s);
86          Console.WriteLine("\nStack[0] = {0}", s[0]);
87          Console.WriteLine("Stack[5] = {0}", s[5]);
88          Console.WriteLine("Stack[14] = {0}\n", s[14]);
89           while (s.Count > 0)
90           Console.WriteLine("Popped {0}", s.Pop());    //выталкивание -(данных) из стека
91          Console.WriteLine("\nStack s = {0}", s);
92          Console.ReadLine();
93      }
94  }
95  }
```

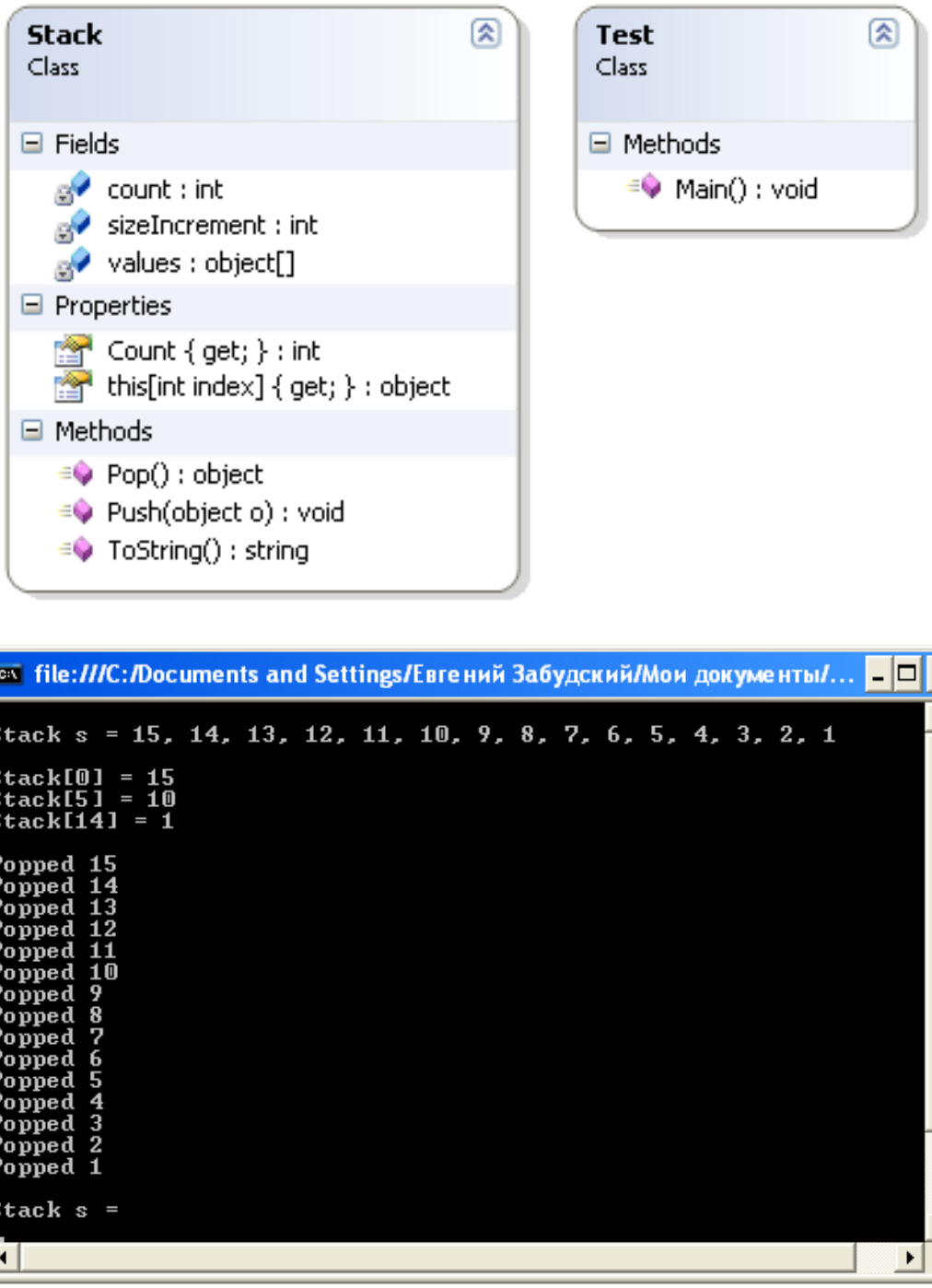

Рис. 2.1. Диаграмма классов и результаты работы программы – Листинг 2.3. Пример компонента-стека

2.6. Перспективы

Размышляя о будущем языка, разработчики всегда испытывают искушение дополнить его новыми функциональными возможностями и сохранить таким же простым, как и прежде. (Этот принцип часто кратко формулируется как "**простота — это тоже функциональная возможность**"). В язык **C#** можно добавить множество новых функциональных возможностей, но только некоторые из них будут внесены в следующие версии. Основное внимание разработчиков языка было сосредоточено на выпуске исходной версии **C#**, поэтому они не могли тратить много времени на обсуждение тех функциональных возможностей, которые можно было бы добавить в будущем.

Одна функциональная возможность, которая, несомненно, пригодилась бы в этом языке, связана с использованием родовых типов (**generic types**), называемых иногда параметризованными ти-

пами (**parameterized types**). В языке **C#** родовые типы используются крайне ограниченно, и это выражается в том, что каждый тип может быть сохранен как объект. Это позволяет создавать классы-коллекции, которые могут содержать любой тип. Такой подход прекрасно работает, но обладает некоторыми нежелательными последствиями, что демонстрируется в приведенном ниже коде.

```
ArrayList  arr  =  new ArrayList();
arr.Add(1);
arr.Add(2);
int value  =  (int)  arr[0];
```

Целые числа **1** и **2** имеют **тип значения**, поэтому они должны быть упакованы (**boxing**) для сохранения в массиве **ArrayList**, который предназначен для сохранения только значений типа **Object**. Для упаковки этих значений потребуется выделить дополнительную память, что связано со снижением общей производительности (это первая проблема).

Второй проблемой является приведение к типу **int** при доступе к этому значению. Данное приведение делает код более неуклюжим, но, что гораздо важнее, во время выполнения должна проводиться проверка, действительно ли данный элемент массива **ArrayList** является целым числом. Иначе говоря, данный код безопасен по отношению к использованию типов во время выполнения, но не во время компиляции.

С помощью **родовых типов** можно было бы справиться с этой ситуацией следующим образом:

```
ArrayList<int>  arr  =  new ArrayList<int>();
arr.Add(1);
arr.Add(2);
int value  = arr[0];
```

В этом случае**: 1)** упаковку (**boxing**) и **2)** приведение типов выполнять не требуется.

2.7. C# и стандартизация

Работа над стандартом языка **C#** при содействии **ECMA** (**E**uropean **C**omputer **M**anufacturers **A**ssociation – Европейская Ассоциация производителей компьютеров) началась в сентябре 2000 года. Помимо компании **Microsoft**, в ней приняли активное участие **Intel, Hewlett-Packard, IBM, Fujitsu, Plum Hall** и многие другие. Техническая часть работы для языка **C#** и инфраструктуры **CLI** (подмножества среды **CLR; C**all **L**evel **I**nterface – прикладной программный интерфейс уровня вызовов) была завершена в **октябре 2001 года**.

В **декабре 2001** года генеральная ассамблея ассоциации **ECMA** проголосовала за одобрение стандартов для языка **C#** и инфраструктуры **CLI**. Кроме того, было решено быстро приступить к одобрению этих стандартов со стороны **ISO**. (В начале **апреля 2003** года язык **C#** и инфраструктура **CLI** сертифицированы Международной организацией по стандартизации /**ISO – I**nternational **O**rganization for **S**tandardization/, т.е. эти разработки **Microsoft** получили статус международных стандартов - **NB**).

Резюме

Первые отклики на язык **C#** были достаточно благоприятными. Большая часть платформы .**NET Framework** создана на **C#**, её разработчики считают этот язык очень удобным для создания такой крупной платформы.
Более подробную информацию о **C#** можно найти в спецификации этого языка по адресу: http://msdn.microsoft.com/net/ecma .
Кроме того, огромное количество информации можно найти по адресу: http://www.gotdotnet.com (есть **русско**язычная версия этого узла — http://www.gotdotnet.ru).

Литература

1. Web-ресурс http://msnet.narod.ru/art/art_001/art_001.htm

2. Забудский Е.И. Учебно-методический комплекс дисциплины «Объектно-ориентированный анализ и программирование». М.: Кафедра АПС ГУ-ВШЭ, 2008.
Web-ресурс – http://new.hse.ru/C7/C17/zabudskiy-e-i/default.aspx , **С. 2**

3. Микелсен К. Язык программирования C#. Лекции и упражнения. Учебник: Пер. с англ./Клаус Микелсен – СПб.: ООО «ДиаСофтЮП», 2002.

4. Уоткинз Д., Хаммонд М, Эйбрамз Б. Программирование на платформе **.NET**.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003.