Министерство образования и науки РФ

Томский государственный университет
систем управления и радиоэлектроники (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

Романенко В.В.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Томск – 2016

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

Целью изучения курса «Объектно-ориентированное программирование» является получение студентами знаний о способах конструирования программ с применением языка C#.

Задачи курса:

• освоить технологию программирования в среде .NET;

• изучить основы языка C# – типы данных, форматирование ввода-вывода, вычисление выражений, операторы ветвления и итераций, создание структур данных, классов, интерфейсов;

• познакомиться с библиотекой классов .NET, изучить основные классы для работы с файлами, потоками и консольными приложениями;

• освоить среду разработки приложений .NET, отладку и документирование кода.

Желательно, чтобы, приступая к изучению данного курса, читатель уже был знаком с навыками объектно-ориентированного программирования на языке C++ или Java. Хотя принципы ООП здесь кратко рассматриваются, язык C# построен на многих заимствованиях из языков C++ и Java, поэтому знание хотя бы одного из них существенно упростит усвоение дисциплины. Также желательно иметь представление об основах программирования в ОС Windows (система сообщений, организация процессов и потоков, API).

В данном учебном пособии предлагаются несколько сред для разработки приложений на языке C#. К пособию прилагаются примеры, основная часть которых выполнена в среде Microsoft Visual Studio 2008, хотя для их компиляции можно использовать и другие среды. Если в тексте пособия встречается следующее обозначение:

 Пример: Samples\2_1_1.

это означает, что примеры, рассмотренные в данном разделе, прилагаются к учебному пособию и располагаются в папке «Samples\2_1_1». В этой папке находится только один проект, его имя совпадает с именем папки, а расширение файла проекта – «.sln». Таким образом, в данном случае файл проекта будет иметь имя «2_1_1.sln». Стандартное имя главного файла с исходным кодом в проекте – «Program.cs». В большинстве проектов это единственный исходный файл. Все исключения из этих правил будут оговариваться отдельно.

Также при рассмотрении синтаксиса различных конструкций, будут использованы условные обозначения в виде квадратных и угловых скобок, а также многоточий. Фигурные скобки используются для группировки параметров. Квадратные скобки означают, что параметр, заключенный в них, является необязательным в данной синтаксической конструкции. Угловые скобки используются для условных обозначений. Т.е. текст, помещенный в них, следует использовать не буквально, а заменять его фактической конструкцией. Многоточие означает повторение предыдущей конструкции (от нуля и более раз). Вертикальная черта – альтернативный синтаксис. Например:

```
{[<имя диска>:][\<имя каталога> ...]\<имя файла>} | nul
{[<имя диска>:][\<имя каталога>[\...]]\<имя файла>} | nul
```

Здесь формально задается спецификация файла. Имя диска и каталога является необязательным. При составлении фактической спецификации обозначение <имя диска> должно быть заменено фактическим именем диска (C, D и т.п.), и т.д. Каталоги могут быть вложенными, что и показано многоточием (приведены два эквивалентных варианта). Вместо имени файла можно указать виртуальный файл nul.

Если фигурные, квадратные или угловые скобки являются частью рассматриваемой конструкции, они будут заключены в кавычки:

```
int"[]" <имя>; // Описание переменной
```

Полужирным шрифтом выделяются ключевые слова языка и директивы препроцессора, но только если они не находятся в области действия комментария или строки. Комментарии выделяются серым цветом.

2. ВВЕДЕНИЕ В ТЕХНОЛОГИЮ .NET

Прежде, чем приступать к изучению технологии Microsoft .NET, познакомимся с принципами объектно-ориентированного программирования (ООП). Предполагается, что читатель уже знаком с основами ООП на других языках (в лучшем случае – C++ или Java), поэтому глубоко погружаться в принципы объектно-ориентированной парадигмы мы не будем. Зная объектно-ориентированный C++, очень легко перейти на C#, в чем-то, как мы увидим в следующих главах, он даже проще.

§ 2.1. Принципы объектно-ориентированного программирования

В этом разделе мы познакомимся с терминологией ООП и убедимся в важности применения в программировании объектно-ориентированных концепций. Бытует мнение, что во многих языках, таких как C++ и Microsoft Visual Basic, есть «поддержка объектов», однако на самом деле лишь немногие из них следуют всем принципам, составляющим основу ООП, и язык C# – один из них. Он изначально разрабатывался как настоящий объектно-ориентированный язык, в основе которого лежит технология компонентов. Взять, например, C++. Своими корнями он глубоко уходит в язык C, и ради поддержки программ, написанных когда-то на процедурном C, в нем пришлось пожертвовать очень многими идеями ООП. Даже в Java есть вещи, не позволяющие считать его по-настоящему объектно-ориентированным языком.

Прежде всего, имеются в виду базисные типы и объекты, которые обрабатываются и ведут себя по-разному. Отметим, что объектно-ориентированное программирование – это не только термин, и не только новый синтаксис или новый интерфейс прикладного программирования (API). ООП – это целый набор концепций и идей, позволяющих осмыслить задачу, стоящую при разработке компьютерной программы, а затем найти путь к ее решению более понятным, а значит, и более эффективным способом.

2.1.1. Понятие объекта и класса

В настоящем объектно-ориентированном языке все элементы так называемой предметной области (problem domain) выражаются через концепцию

объектов. Объекты – это центральная идея объектно-ориентированного программирования. Если мы обдумываем стоящую перед нами проблему, то не оперируем понятиями «структура», «пакет данных», «вызов функций» и «указатели», ведь привычнее применять понятие «объектов».

Например, мы пишем программу приложение для выписки счета-фактуры, в котором нужно подсчитать сумму по всем позициям. Рассмотрим две формулировки:

• Не объектно-ориентированный подход: заголовок счета-фактуры представляет структуру данных, к которой мы получим доступ. В эту структуру войдет также динамический список структур, содержащих описание и стоимость каждой позиции. Поэтому для получения общего итога по счету нам потребуется объявить переменную с именем наподобие totalAmount и инициализировать ее нулем, получить указатель на структуру-заголовок счета, получить указатель на начало динамического списка, а затем «пробежать» по всему этому списку. Просматривая элементы списка для каждой позиции, мы будем брать оттуда переменную-член, где находится итог для данной позиции, и прибавлять его к totalAmount.

• Объектно-ориентированный подход: у нас будет объект «счет-фактура», и ему мы отправим сообщение с запросом на получение общей суммы. При этом не важно, как информация хранится внутри объекта, как это было в предыдущем случае. Мы общаемся с объектом естественным образом, запрашивая у него информацию посредством сообщений (группа сообщений, которую объект в состоянии обработать, называется интерфейсом объекта).

Очевидно, что объектно-ориентированный подход естественнее и ближе к тому способу рассуждений, которым мы руководствуемся при решении задач. Во втором варианте объект «счет-фактура», наверно, просматривает в цикле совокупность (collection) объектов, представляющих данные по каждой позиции, посылая им запросы на получение суммы по данной позиции. Но если требуется получить только общий итог, то нам все равно, как это реализовано, так как одним из основных принципов объектно-ориентированного программирования является инкапсуляция (encapsulation). Инкапсуляция – это свойство объекта скрывать свои внутренние данные и методы, представляя наружу только интерфейс, через который осуществляется программный доступ к самым важным элементам объекта. Как объект

выполняет задачу, не имеет значения, главное, чтобы он справлялся со своей работой. Имея в своем распоряжении интерфейс объекта, мы заставляем объект выполнять нужную нам работу.

Во втором подходе от объекта требовалось, чтобы он произвел нужную нам работу, т.е. подсчитал общий итог. В отличие от структуры, в объект по определению входят не только данные, но и методы их обработки. Это значит, что при работе с некоторой проблемной областью можно не только создать нужные структуры данных, но и решить, какие методы связать с данным объектом, чтобы объект стал полностью инкапсулированной частью функциональности системы.

Допустим, мы пишем приложение для расчета зарплаты служащим нашей компании. Код на C, представляющий данные о служащем, будет выглядеть примерно так:

```
struct EMPLOYEE
{
      char szName[25];
      int iAge;
      double dPayRate;
};
```

А код для расчета зарплаты, например, Анны, в котором используется структура EMPLOYEE, выглядит так:

```
void main()
{
      double dTotalPay;
      struct EMPLOYEE *pEmp;
      pEmp = (EMPLOYEE *)malloc(sizeof(EMPLOYEE));
      if (pEmp)
      {
            pEmp->dPayRate = 100;
            strcpy(pEmp->szName, "Анна");
            pEmp->iAge = 28;
            dTotalPay = pEmp->dPayRate * 40;
            printf("Зарплата Анны составляет %g\n", dTotalPay);
      }
      free(pEmp);
}
```

Код этого примера основан на данных, содержащихся в структуре, и на некотором внешнем (по отношению к структуре) коде, обрабатывающем эту структуру. И что же здесь не так? Основной недостаток – в отсутствии абстрагирования: при работе со структурой EMPLOYEE необходимо знать чересчур много о данных, описывающих служащего. Почему это плохо? Допустим, спустя какое-то время нам потребуется определить «чистую» зарплату

Анны (после удержания всех налогов). Тогда пришлось бы не только изменить всю клиентскую часть кода, работающую со структурой EMPLOYEE, но и составить описание (для других программистов, которым может достаться этот код впоследствии) изменений в функционировании программы.

Теперь рассмотрим тот же пример на C#:

```
using System;

class Employee
{
	public Employee(string name, int age, double rate)
	{
		Name = name;
		Age = age;
		PayRate = rate;
	}

	private string Name;
	private int Age;
	private double PayRate;

	public double CalculatePay(int hours)
	{
		// Здесь вычисляется зарплата
		return PayRate * hours;
	}
}

class EmployeeApp
{
	public static void Main()
	{
		Employee emp = new Employee("Анна", 28, 100);
		Console.WriteLine("Зарплата Анны составляет " +
			emp.CalculatePay(40));
	}
}
```

В C#-версии примера пользователю объекта для вычисления зарплаты достаточно вызвать его метод CalculatePay. Преимущество этого подхода в том, что пользователю больше не нужно следить, как рассчитывается зарплата. Если когда-нибудь потребуется изменить способ ее вычисления, то эта модификация не скажется на существующем коде. Такой уровень абстрагирования – одно из основных преимуществ использования объектов.

Конечно, в клиентской части кода на языке C можно создать функцию доступа к структуре EMPLOYEE., которая будет вычислять зарплату. Однако ее придется создавать отдельно от структуры, которую она обрабатывает, и мы окажемся перед той же проблемой. А вот в объектно-ориентированном языке вроде C# данные объекта и методы их обработки (интерфейс объекта)

всегда будут вместе.

Модифицировать переменные объекта следует только методами этого же объекта. Как видно из нашего примера, все переменные-члены в классе Employee объявлены с модификатором доступа **private**, а метод CalculatePay – с модификатором **public**. Модификаторы доступа применяются для задания уровня доступа к членам класса. Модификатор **private** указывает, что доступ к члену имеет только сам класс, а клиентский код – нет. Модификатор **public** делает член доступным как для любых классов, так и для клиентского кода. Подробнее о модификаторах доступа поговорим в § 4.2.

 Пример: Samples\2.1\2_1_1.

2.1.1.1. Объект или класс?

Программисты, начинающие осваивать ООП, часто путают термины «объект» и «класс».

Есть разные трактовки термина «класс», показывающие, в частности, чем класс отличается от объекта. Будем считать, что *класс* – это просто новый тип данных (как **char**, **int** или **long**), с которым связаны некие данные и методы для их обработки. Объект же – это экземпляр типа, или класса.

Можно понимать класс как «чертеж» объекта. Разработчик объекта сначала создает его «чертеж», так же, как инженер-строитель сначала чертит план дома. Имея такой чертеж, мы располагаем всего лишь проектом дома этого типа. Однако те, кто приобрел этот чертеж, могут по нему построить себе дом. Таким же образом на базе класса – «чертежа» набора функциональных возможностей – можно создать объект, обладающий всеми возможностями этого класса.

2.1.1.2. Реализация

Реализация (instantiation) в ООП означает факт создания экземпляра (он же объект) некоторого класса. В следующем примере мы создадим только класс, или спецификацию (specification), объекта. А поскольку это не сам объект, а лишь его «чертеж», то память для него не выделяется.

Чтобы получить объект класса и начать с ним работу, мы должны создать экземпляр класса в своем методе примерно так:

```
class EmployeeApp
{
```

```
        public static void Main()
        {
            Employee emp = new Employee("Анна", 28, 100);
        }
    }
```

В этом примере объявлена переменная «emp» типа Employee, и с помощью оператора **new** выполнена ее реализация. Переменная «emp» представляет собой экземпляр класса Employee и является объектом Employee. Выполнив реализацию объекта, мы можем установить с ним связь через его открытые (**public**) члены. Например, для объекта «emp» это метод CalculatePay. Пока реально объект не существует, вызывать его методы нельзя (за исключением статических членов, о которых мы поговорим в главе 4). Посмотрим на следующий код C#:

```
    class EmployeeApp
    {
        public static void Main()
        {
            Employee emp = new Employee("Анна", 28, 100);
            Employee emp2 = new Employee("Яна", 34, 120);
        }
    }
```

Здесь два экземпляра одного класса Employee – «emp» и «emp2». Оба объекта одинаковы с точки зрения программной реализации, но у каждого экземпляра свой набор данных, который может обрабатываться отдельно от другого.

2.1.2. Три основных принципа ООП

По Бьерну Страуструпу, автору языка C++, язык может называться объектно-ориентированным, если в нем реализованы три концепции: объекты, классы и наследование. Однако теперь принято считать, что такие языки должны держаться на других трех китах: инкапсуляции, наследовании и полиморфизме. Этот философский сдвиг произошел из-за того, что со временем мы стали понимать: построить объектно-ориентированные системы без инкапсуляции и полиморфизма так же невозможно, как без классов и наследования.

 Пример: Samples\2.1\2_1_2.

2.1.2.1. Инкапсуляция

Как уже было сказано выше, *инкапсуляция*, или утаивание информации

(information hiding), – это возможность скрыть внутреннее устройство объекта от его пользователей, предоставив через интерфейс доступ только к тем членам объекта, с которыми клиенту разрешается работать напрямую. Поскольку в том же контексте мы говорили также об *абстрагировании*, то рассмотрим разницу между этими похожими понятиями. Инкапсуляция подразумевает наличие границы между внешним интерфейсом класса (открытыми членами, видимыми пользователям класса) и деталями его внутренней реализации. Преимущество инкапсуляции для разработчика в том, что он может открыть те члены класса, которые будут оставаться статичными, или неизменяемыми, скрыв внутреннюю организацию класса, более динамичную и в большей степени подверженную изменениям. Как уже говорилось, в C# инкапсуляция достигается путем назначения каждому члену класса своего модификатора доступа.

Абстрагирование связано с тем, как данная проблема представлена в пространстве программы. Во-первых, абстрагирование заложено в самих языках программирования. Программируя на языках высокого уровня, нам не приходится заботиться о стеке или регистрах процессора. Причина в том, что большинство языков отстраняют нас (абстрагируют) от таких подробностей, позволяя сосредоточиться на решении прикладной задачи.

При программировании на объектно-ориентированных языках скрываются элементы, не связанных напрямую с решением задачи, позволяя нам полностью сосредоточиться на самой задаче и решить ее более эффективно.

Однако язык – это один уровень абстрагирования. Если мы пойдем дальше, то, как разработчикам класса, нам нужно придумать такую степень абстрагирования, чтобы клиенты нашего класса могли сразу сосредоточиться на своей задаче, не тратя время на изучение работы класса. На вопрос – какое отношение интерфейс класса имеет к абстрагированию? – можно ответить так: интерфейс класса и есть реализация абстрагирования.

Поясним это по аналогии с работой внутренних устройств торговых автоматов. Описать подробно, что происходит внутри торгового автомата, довольно трудно. Чтобы выполнить свою задачу, автомат должен принять деньги, рассчитать, дать сдачу, а затем – требуемый товар. Однако покупателям – пользователям автомата – видно лишь несколько его функций. Элементы интерфейса автомата: щель для приема денег, кнопки выбора товара, рычаг для запроса сдачи, лоток, куда поступает сдача, и желоб подачи товара.

Торговые автоматы остаются без изменений (более или менее) со времени их изобретения. Это связано с тем, что их внутренняя организация совершенствовалась по мере развития технологии, а основной интерфейс не нуждался в больших переменах. Неотъемлемой частью проектирования интерфейса класса является достаточно глубокое понимание предметной области. Такое понимание поможет нам создать интерфейс, предоставляющий пользователям доступ к нужной им информации и методам, но изолирующий их от «внутренних органов» класса. При разработке интерфейса мы должны думать не только о решении текущей задачи, но и о том, чтобы обеспечить такое абстрагирование от внутреннего представления класса, которое позволит неограниченно модифицировать закрытые члены класса, не затрагивая существующего кода. Кроме того, решая, какие члены класса сделать открытыми, надо опять вспомнить о клиенте. При разработке своих классов мы всегда должны ставить себя на место программиста, которому предстоит работать либо с экземплярами наших классов, либо с производными от них классами.

2.1.2.2. Наследование

Наследованием называют возможность при описании класса указывать на его происхождение (kind-of relationship) от другого класса. Наследование позволяет создать новый класс, в основу которого положен существующий. В полученный таким образом класс можно внести свои изменения, а затем создать новые объекты данного типа. Этот механизм лежит в основе создания так называемой *иерархии* классов. После абстрагирования наследование – наиболее значимая часть общего планирования системы. Производным (derived class) называется создаваемый класс, производный от базового класса (base class). Производный класс наследует все члены базового класса. Какие именно члены базового класса наследуются производными классами, решается в C# через модификаторы доступа.

Чтобы понять, когда и как применять наследование, вспомним пример EmployeeApp. Допустим, в компании есть служащие с разными типами оплаты труда: постоянный оклад (salaried), почасовая оплата (hourly) и оплата по договору (contract). Хотя у всех объектов Employee должен быть одинаковый интерфейс, их внутреннее функционирование может различаться. Например, метод CalculatePay для служащего на окладе будет работать не так, как для контрактника. Однако для наших пользователей важно, чтобы интерфейс

CalculatePay не зависел от того, как считается зарплата.

Нельзя ли здесь обойтись без объектов? Введем в структуру EMPLOYEE член, описывающий тип оплаты, и напишем следующую функцию:

```
double CalculatePay(EMPLOYEE *emp, int hours)
{
      double dTotalPay;
      // Проверяем тип оклада
      if (emp->type == SALARIED)
      {
            // Вычисляем заработок для служащего на окладе
      }
      else if (emp->type == CONTRACT)
      {
            // Вычисляем заработок по контракту
      }
      else if (emp->type == HOURLY)
      {
            // Вычисляем почасовой заработок
      }
      else
      {
            // Выполняем иную обработку
      }
      return dTotalPay;
}
```

В этом коде есть две проблемы. Во-первых, успешное выполнение функции тесно связано со структурой EMPLOYEE. Как уже было сказано, подобная связь очень нежелательна, поскольку любое изменение структуры потребует модификации этого кода. Т.е. мы опять должны нагружать пользователей нашего класса подробностями, в которых непосвященному разобраться трудно. Это все равно, как если бы производитель автомата по продаже газировки, перед тем как выдать стакан воды, потребовал бы от покупателя знаний о работе внутренних механизмов автомата.

Во-вторых, такой код нельзя задействовать повторно. Тот, кто понимает, что наследование способствует повторному использованию кода, теперь по достоинству оценит классы и объекты. Так, в нашем примере достаточно описать в базовом классе те члены, которые будут функционировать независимо от типа оплаты, а любой производный класс унаследует функции базового класса, добавив к ним что-то свое. Так это выглядит на C#:

```
using System;

class Employee
{
      public Employee(string id, string name, int age, double rate)
```

```
        {
            EmployeeId = id;
            Name = name;
            Age = age;
            PayRate = rate;
        }

        protected string EmployeeId;
        protected string Name;
        protected int Age;
        protected double PayRate;
    }

    class SalariedEmployee : Employee
    {
        public string PensionNumber;

        public SalariedEmployee(string id, string name, int age,
            double rate, string pension) : base(id, name, age, rate)
        {
            PensionNumber = pension;
        }

        public double CalculatePay(int days)
        {
            // Вычисляем заработок постоянного служащего
            // из расчета 8 рабочих часов в день
        }
    }

    class ContractEmployee : Employee
    {
        public double ContractHours;

        public ContractEmployee(string id, string name, int age,
            double rate, double hours) : base(id, name, age, rate)
        {
            ContractHours = hours;
        }

        public double CalculatePay(int percent)
        {
            // Вычисляем заработок для контрактника
            // исходя из доли выполненной работы
        }
    }

    class HourlyEmployee : Employee
    {
        public HourlyEmployee(string id, string name, int age,
            double rate) : base(id, name, age, rate)
        {
        }

        public double CalculatePay(int hours)
        {
            // Вычисляем заработок для почасового служащего
        }
    }
```

Отметим важные моменты, вытекающие из данного примера:

• В базовом классе Employee описана строковая переменная EmployeeId, которая наследуется и классом SalariedEmployee, классом ContractEmployee и классом HourlyEmployee. Все производные классы получили эту переменную автоматически как наследники класса Employee.

• Каждый из производных классов реализует свою версию CalculatePay. Хотя реализация этих функций различна, пользовательский код останется прежним. Базовый класс этой функции лишен, т.к. если не известен тип оплаты труда, то провести расчет невозможно.

• Производные классы в дополнение к членам, унаследованным из базового класса, имеют свои члены: в классе SalariedEmployee описана строковая переменная PensionNumber (номер пенсионного страхового свидетельства), а в класс ContractEmployee включено описание члена ContractHours (количество часов, прописанное в контракте).

• Мы изменили модификатор доступа у полей класса с **private** на **protected**. Сделано это для того, чтобы производные классы могли получить доступ к этим полям. Внешний клиентский код доступа к ним по-прежнему не имеет.

• Конструкторы производных классов явно вызывают конструктор класса базового.

Этот небольшой пример показывает, как наследование функциональных возможностей базовых классов позволяет создать повторно используемый код. Кроме того, мы можете расширить эти возможности, добавив собственные переменные и методы.

2.1.2.3. Полиморфизм

Полиморфизм – это функциональная возможность, позволяющая старому коду вызвать новый код. Это свойство ООП наиболее ценно, поскольку дает нам возможность расширять и совершенствовать свою систему, не затрагивая существующий код.

Предположим, нам нужно написать метод, в котором для каждого объекта из набора Employee вызывается метод CalculatePay. Все просто, если зарплата рассчитывается одним способом: мы можем сразу вставить в набор тип нужного объекта. Проблемы начинаются с появлением других форм оплаты – в этом случае опять придется переписывать много кода. Объектно-

ориентированный язык решает эту проблему благодаря полиморфизму.

В нашем примере надо описать *абстрактный* базовый класс Employee, а затем создать производные от него классы для всех форм оплаты (упомянутых выше). В базовом классе будет находиться абстрактный же метод CalculatePay (т.е. не имеющий тела – мы не можем вычислить оплату, не зная ее формы). Каждый производный класс будет иметь собственную реализацию метода CalculatePay. При вычислении оплаты необходимо привести его к типу класса-предка и вызвать *виртуальный перегруженный* метод этого объекта, а средства языка времени выполнения обеспечат нам, благодаря полиморфизму, вызов той версии этого метода, которая нам требуется. Поясним сказанное на примере:

```
using System;

abstract class Employee
{
      public Employee(string id, string name, int age, double rate)
      {
            EmployeeId = id;
            Name = name;
            Age = age;
            PayRate = rate;
      }

      protected string EmployeeId;
      protected string Name;
      protected int Age;
      protected double PayRate;

      public abstract double CalculatePay(int param);

      public string AName
      {
            get { return Name; }
      }
}

class SalariedEmployee : Employee
{
      public string PensionNumber;

      public SalariedEmployee(string id, string name, int age,
            double rate, string pension) : base(id, name, age, rate)
      {
            PensionNumber = pension;
      }

      public override double CalculatePay(int days)
      {
            // Вычисляем заработок постоянного служащего
            // из расчета 8 рабочих часов в день
            Console.Write(" :: SalariedEmployee.CalculatePay :: ");
            return PayRate * days * 8;
```

```
        }
    }

    class ContractEmployee : Employee
    {
        public double ContractHours;

        public ContractEmployee(string id, string name, int age,
            double rate, double hours) : base(id, name, age, rate)
        {
            ContractHours = hours;
        }

        public override double CalculatePay(int percent)
        {
            // Вычисляем заработок для контрактника
            // исходя из доли выполненной работы
            Console.Write(" :: ContractEmployee.CalculatePay :: ");
            return ContractHours * PayRate * percent / 100;
        }
    }

    class HourlyEmployee : Employee
    {
        public HourlyEmployee(string id, string name, int age,
            double rate) : base(id, name, age, rate)
        {
        }

        public override double CalculatePay(int hours)
        {
            // Вычисляем заработок для почасового служащего
            Console.Write(" :: HourlyEmployee.CalculatePay :: ");
            return PayRate * hours;
        }
    }

    class EmployeeApp
    {
        private static Employee[] employees;

        private static void CalculatePay()
        {
            foreach(Employee emp in employees)
            {
                double pay;
                Console.Write("Зарплата служащего " + emp.AName +
                    " составляет");
                pay = emp.CalculatePay(40);
                Console.WriteLine(pay);
            }
        }

        public static void Main()
        {
            employees = new Employee[3];
            employees[0] = new SalariedEmployee("111-111-111",
                "Анна", 28, 100, "555-555-555-55");
            employees[1] = new ContractEmployee("111-111-112",
                "Яна", 34, 120, 100);
```

```
                employees[2] = new HourlyEmployee("111-111-113",
                      "Инна", 24, 90);
                CalculatePay();
            }
      }
```

В результате компиляции и запуска этого приложения будут получены такие результаты:

```
      Зарплата служащего Анна составляет :: SalariedEmployee.CalculatePay ::
32000
      Зарплата служащего Яна составляет :: ContractEmployee.CalculatePay ::
4800
      Зарплата служащего Инна составляет :: HourlyEmployee.CalculatePay ::
3600
```

Полиморфизм имеет, как минимум, два плюса. Во-первых, он позволяет группировать объекты, имеющие общий базовый класс, и последовательно (например, в цикле) их обрабатывать. В рассмотренном случае у нас три разных типа объектов (SalariedEmployee, ContractorEmployee и HourlyEmployee), но мы вправе считать их все объектами Employee, поскольку они произведены от базового класса Employee. Поэтому их можно поместить в массив, описанный как массив объектов Employee. Во время выполнения вызов метода одного из этих объектов будет преобразован, благодаря полиморфизму, в вызов метода соответствующего производного объекта.

Второе достоинство уже упоминалось – старый код может использовать новый код. Заметьте: метод EmployeeApp.CalculatePay перебирает в цикле элементы массива объектов Employee. Поскольку объекты приводятся неявно к вышестоящему типу Employee, а реализация полиморфизма во время выполнения обеспечивает вызов надлежащего метода, то ничто не мешает нам добавить в систему другие производные формы оплаты, вставить их в массив объектов Employee, и весь существующий код будет работать, не потребовав модификации.

§ 2.2. Технология Microsoft .NET

Не имея четкого представления о Microsoft .NET и роли, которую играет в этой новой инициативе Microsoft язык C#, нам будет трудно разобраться в ключевых элементах C#, поддерживаемых платформой Microsoft .NET. Представленный в этом разделе обзор технологии Microsoft .NET поможет нам усвоить терминологию и понять, почему некоторые элементы языка C# ведут себя так, а не иначе. Если просмотреть в Интернете материалы по Microsoft .NET, можно заметить разнобой в трактовке и употреблении терминов этой технологии. Двусмысленные, а порой и просто противоречивые высказывания мешают уловить суть излагаемого материала. Во многом это объясняется новизной проблемы. Поэтому первым делом мы постараемся разобраться с этим и разъяснить некоторые термины, связанные с Microsoft .NET.

2.2.1. Платформа Microsoft .NET

Основу Microsoft .NET составляют четыре базовых компонента:

• .NET Building Block Services – средства программного доступа к таким службам, как хранилище файлов (file storage), календарь (calendar), служба аутентификации «Passport.NET»;

• ПО для устройств .NET, которое будет выполняться на новых устройствах Интернета;

• Средства .NET для работы с пользователями, включающие естественный интерфейс (natural interface), информационные агенты (information agents) и интеллектуальные теги (smart tags) – технологию, которая автоматизирует переход по гиперссылкам к информации, связанной со словами и фразами в документах пользователей;

• Инфраструктура .NET, состоящая из .NET Framework, Microsoft Visual Studio .NET, .NET Enterprise Servers и Microsoft Windows .NET.

Большинство разработчиков воспринимает инфраструктуру .NET собственно как .NET. Поэтому в дальнейшем при любом упоминании .NET (если нет предварительной оговорки) мы будем иметь в виду инфраструктуру .NET. Инфраструктура .NET связана со всеми технологиями, составляющими новую среду создания и выполнения надежных, масштабируемых, распределенных приложений. Та часть .NET, с помощью которой разрабатываются такие приложения, называется .NET Framework.

Технология .NET Framework состоит из Common Language Runtime (CLR) и набора библиотек классов .NET Framework, который иногда называют Base Class Library (BCL). Среда CLR – это, по сути, виртуальная машина, в которой функционируют приложения .NET. Все языки .NET имеют в своем распоряжении библиотеки классов .NET Framework. Если вы знакомы с Microsoft Foundation Classes (MFC), либо Object Windows Library (OWL) и Visual Component Library (VCL) компании Borland, то вам не надо объяснять, что это такое. Библиотеки классов .NET Framework включают поддержку практически всех технологий. Вообще, библиотеки классов .NET Framework столь обширны, что даже поверхностный обзор всех поддерживаемых классов потребует отдельной книги.

Под термином «виртуальная машина» здесь не подразумевается Java Virtual Machine (JVM). Здесь применяется этот термин в его традиционном значении. Несколько десятилетий назад IBM ввела в оборот словосочетание «виртуальная машина» («virtual machine»). Этим термином была обозначена абстракция высокоуровневой ОС, внутри которой могли функционировать в полностью инкапсулированной среде другие ОС. Говоря о CLR как о виртуальной машине, имеется в виду то, что код, выполняемый в инкапсулированной и управляемой среде, отделен от других процессов на этой машине.

2.2.2. Common Language Runtime

Среда Common Language Runtime (CLR) – это сердце технологии Microsoft .NET. Как следует из названия, это среда времени выполнения кода, в которой обеспечивается эффективное взаимодействие приложений, пересекающее границы разных языков программирования (cross-language interoperability). Как достигается это взаимодействие? При помощи Common Language Specification (CLS) – набора правил, которых должен придерживаться компилятор языка при создании .NET-приложений, запускаемых в среде CLR. Любой, кто захочет написать компилятор для .NET, должен следовать этим правилам, и тогда приложения, сгенерированные этим компилятором, будут работать наряду с другими .NET-приложениями и будут иметь такую же возможность взаимодействия.

С CLR связана важная концепция управляемого кода (managed code) – кода, выполняемого только в среде CLR и управляемого ею. Напомню, что во время исполнения в нынешних ОС Microsoft Windows мы имеем дело с раз-

нородными независимыми друг от друга процессами. Единственное требование, которому должны отвечать приложения в среде Windows, состоит в том, чтобы они правильно работали. Эти приложения создаются совершенно разными компиляторами. Иначе говоря, приложения должны подчиняться только наиболее общим правилам работы под Windows.

В среде Windows есть несколько глобальных правил поведения приложений, относящихся к их взаимодействию друг с другом, распределению памяти, а также к привлечению средств самой ОС для работы от их имени. Напротив, в среде управляемого кода есть набор правил, обеспечивающих единообразное в глобальном смысле поведение всех приложений независимо от того, на каком языке они написаны. Единообразное поведение .NET-приложений – характерная черта технологии .NET, и его нельзя игнорировать. К счастью, эти глобальные правила распространяются главным образом только на создателей компиляторов.

2.2.3. Библиотеки классов .NET Framework

Библиотеки классов .NET Framework играют чрезвычайно важную роль в обеспечении межъязыкового взаимодействия приложений, так как они позволяют разработчикам использовать единый программный интерфейс ко всем функциональным средствам CLR. Библиотеки классов .NET Framework делают фактически революционный прорыв в разработке компиляторов. До .NET почти каждый автор компилятора разрабатывал язык, обладающий способностью делать большую часть своей собственной работы. Даже C++, разработанный как набор функциональных возможностей, работающих совместно с библиотекой классов, имеет некоторые средства для собственных нужд. Тогда как роль языков в окружении .NET не исчерпывается предоставлением синтаксических интерфейсов к библиотекам классов .NET Framework.

В качестве иллюстрации к сказанному сравним версии традиционного приложения «Hello, World!» на языках C++ и C#. Код на языке C++:

```
#include <iostream>

int main(int argc, char* argv[])
{
      cout << "Hello, World!" << endl;
      return 0;
}
```

В начало приложения включен заголовочный файл с объявлением функции класса ostream (экземпляром которого является cout). Функция main – входная точка любого приложения на C/C++ – выводит на стандартное устройство вывода с помощью объекта cout строку «Hello, World!». Здесь для нас важно то, что написать такое приложение на языке .NET без библиотек классов .NET Framework нельзя. Это действительно так: в .NET-языках нет присущих обычным компиляторам основных элементов, которые, например, выводят на консоль строку текста. Да, с точки зрения технологии, реализация объекта cout находится в той части C/C++, которая сама является библиотекой. И все-таки основные задачи C++, такие как форматирование строк, файловый ввод-вывод и вывод на экран, хотя бы формально считаются частью исходного языка. Что касается C# (и это характерно для любого .NET-языка), то он не в состоянии выполнить даже самую примитивную задачу без привлечения библиотеки классов .NET Framework. А так выглядит пример «Hello, World!» на языке C#:

```
using System;

class Program
{
      public static int Main(string[] args)
      {
            Console.WriteLine("Hello, World!");
            return 0;
      }
}
```

Подробный разбор этой программы, а также сведения о выборе среды разработки приведены в § 2.3.

Итак, что дает этот стандартный набор библиотек классов и чем он хорош? Это зависит от точки зрения. Благодаря такому набору все языки .NET в идеале располагают одними и теми же функциональными возможностями, поскольку все они могут что-то делать (если речь идет не только об объявлении переменных) только с помощью этих библиотек.

В идеале библиотеки классов .NET Framework открывают пользователям языка все функциональные возможности CLR, однако на самом деле так бывает не всегда. Камнем преткновения между разработчиками библиотек классов .NET Framework и разработчиками компиляторов является то, что первые, хотя и попытались открыть для любых языков все функциональные возможности библиотек классов, последних все-таки ничто не обязывает делать реализацию каждой такой возможности (не пренебрегая, правда, мини-

мальными стандартами CLS). Поэтому вряд ли каждый язык .NET будет иметь доступ ко всем функциональным возможностям .NET Framework, поскольку каждая бригада разработчиков компиляторов вправе реализовать только те, что они считают самыми нужными для своих пользователей. Однако, C# – по-видимому, тот язык, в котором имеется доступ практически ко всем функциональным возможностям .NET Framework.

2.2.4. Microsoft Intermediate Language и компиляторы JITter

Для облегчения перевода языков в среду .NET в Microsoft разработан промежуточный язык – Microsoft Intermediate Language (MSIL). Чтобы откомпилировать приложение для .NET, компиляторы берут исходный код и создают из него MSIL-код. В целом, MSIL – это полноценный язык, пригодный для написания приложений. Однако, как в случае с ассемблерным языком, вам вряд ли придется этим заниматься, кроме каких-то особых обстоятельств. Каждая группа разработчиков компилятора решает, в какой мере он будет поддерживать MSIL. Но если создатели компиляторов захотят, чтобы их язык полноценно взаимодействовал с другими языками, им придется ограничить себя рамками, определяемыми спецификациями CLS.

Результатом компиляции приложения, написанного на C# или другом языке, который отвечает правилам CLS, является MSIL-код. Потом, при первом запуске приложения в среде CLR, MSIL-код компилируется в машинные команды, специфичные для данного процессора (рис. 2.1).

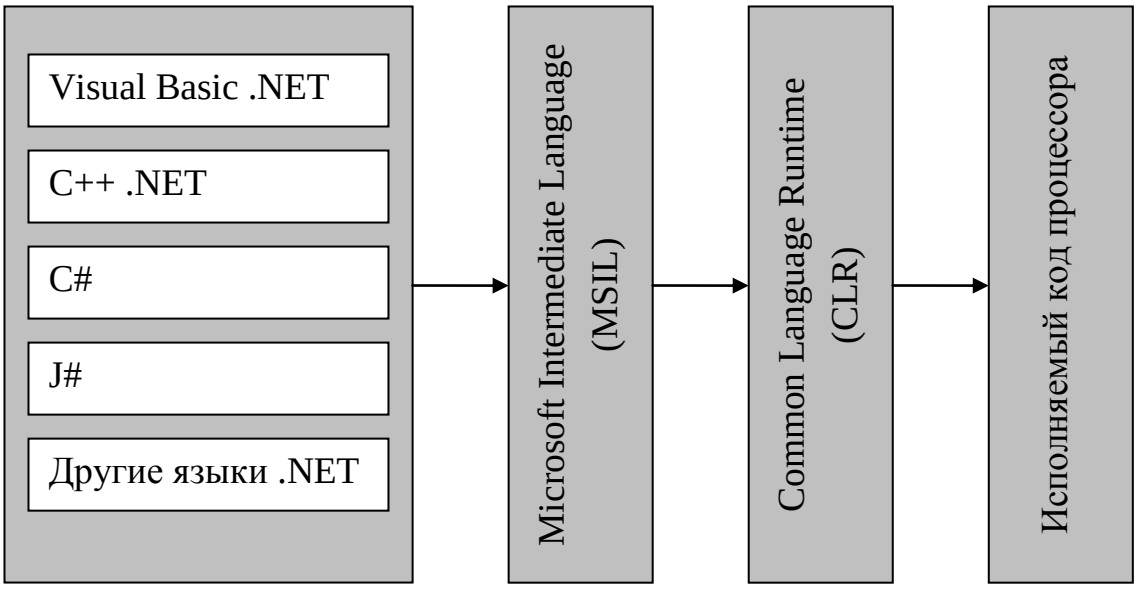


Рис. 2.1 – Компиляция приложения в среде .NET

Посмотрим по порядку, что же происходит с кодом:

1. Мы пишем исходный код на C#.

2. Затем компилируем его с помощью компилятора языка C# в EXE-файл.

3. Компилятор создает MSIL-код и помещает в раздел «только-на-чтение» выходного файла стандартный заголовок PE (признак машинно-независимой выполняемой программы для Win32). Здесь появляется очень важная деталь: при создании выходного файла компилятор импортирует из CLR функцию _CorExeMain.

4. Когда приложение начинает выполняться, ОС загружает этот PE (впрочем, как и обычный PE), а также все нужные DLL, в частности, библиотеку, которая экспортирует функцию _CorExeMain (mscoree.dll).

5. Загрузчик ОС выполняет переход в точку входа PE, устанавливаемую компилятором. Это ничем не отличается от процедуры загрузки в Windows любого другого PE. Однако, так как ОС не в состоянии выполнить MSIL-код, то фактически в точке входа содержится заглушка, в которой установлена команда перехода к функции _CorExeMain из mscoree.dll.

6. Функция _CorExeMain переходит к выполнению MSIL-кода, помещенного в секцию PE.

7. Так как MSIL-код не может быть выполнен непосредственно (ведь это не машинный код), CLR компилирует его с помощью оперативного (just-in-time, или JIT) компилятора (его еще называют JITter) в команды процессора. Эта компиляция выполняется только для непосредственно вызываемых методов программы. Откомпилированный выполняемый код сохраняется на машине и перекомпилируется только в случае изменения исходного кода. Для преобразования MSIL в настоящий машинный код можно применить один из следующих JIT-компиляторов:

1) Генератор кода при установке (install-time code generation). Выполняет компиляцию всей сборки в двоичный код, специфичный для данного процессора, подобно тому, как это делает компилятор C#. *Сборка* (assembly) – это комплект модулей кода, посылаемый компилятору. Эта компиляция выполняется в ходе установки, когда обработка сборки JIT-компилятором меньше всего заметна для конечного пользователя. Достоинство этого типа генерации двоичного кода в том, что компиляция всей сборки выполняется один раз еще до запуска приложения. Поскольку код скомпилирован, то о

потере производительности при первом вызове метода приложения можно не беспокоиться. Вопрос о целесообразности применения этой утилиты решается в зависимости от размера конкретной системы и среды, в которой происходит ее развертывание. Если вы, как это часто бывает, собираетесь создать для своей системы установочное приложение, используйте этот JIT-компилятор, чтобы у пользователя была уже полностью оптимизированная версия системы.

2) Компилятор JIT – стандартный JITter, вызываемый при выполнении приложения каждый раз для впервые активизируемого метода (в порядке, описанном выше). Данный режим работает по умолчанию, если вы не запускаете явно компилятор PreJIT.

3) Компилятор EconoJIT – включается во время выполнения приложения и предназначен специально для систем, которые имеют ограниченные ресурсы, например, для портативных устройств с малым размером памяти. Основное отличие этого компилятора от обычного компилятора JITter – в объединении кодовых фрагментов (code pitching). Благодаря разбивке кода на фрагменты EconoJIT может удалить сгенерированный (т.е. откомпилированный) код, если памяти для запуска системы недостаточно. Достоинство этого компилятора в экономии памяти, а недостаток в том, что если фрагментированный код загружается вновь, он должен быть опять перекомпилирован как код, который еще никогда не вызывался.

2.2.5. Унифицированная система типов

Одна из ключевых черт любой среды разработки – ее система типов. Если среда разработки имеет небольшой выбор типов или ограничивает возможность программиста добавлять свои типы, то такая среда проживет недолго. Среда CLR не только предоставляет разработчику единую унифицированную систему типов CTS (Common Type System), доступную для всех CLS-совместимых языков, но и позволяет создателям языков расширять систему типов путем добавления новых типов, по виду и поведению ничем не отличающихся от встроенных типов. Это означает, что разработчик может работать со всеми типами единообразно независимо от того, стандартные это типы или вновь созданные.

2.2.5.1. Преимущества использования CTS

Одна из ключевых функций любого языка или среды периода выполнения – поддержка типов. Однако наличие унифицированной системы типов дает также множество других выгод.

Возможность взаимодействия языков. Система CTS играет важную роль в обеспечении способности к взаимодействию языков, так как она определяет набор типов, которые должен поддерживать компилятор .NET, чтобы обеспечивать взаимодействие с другими языками. Сама CTS определена в спецификации CLS (Common Language Specification). Система CLS определяет единый набор правил для каждого компилятора .NET, гарантируя, что каждый компилятор будет выдавать код, согласованно взаимодействующий с CLR. Согласно требованиям CLS, компилятор должен поддерживать некоторые типы, определенные в CTS. Все языки .NET используют единую систему типов. Это выгодно, так как гарантирует бесшовное взаимодействие объектов и типов, создаваемых на различных языках. Благодаря этой комбинации CTS/CLS взаимодействие программ на разных языках становится реальным.

Иерархия объектов с единым корнем. Итак, важная характеристика CTS – иерархия объектов с единым корнем. В .NET Framework каждый тип системы происходит от базового класса System.Object. Подход, использующий единый базовый класс – важное отличие от языка C++, в котором нет базового класса для всех классов. Он рекомендован теоретиками ООП и реализован в большинстве объектно-ориентированных языков, формирующих главное направление этой технологии.

Иерархия объектов с единым корнем является ключевой для унифицированной системы типов, поскольку гарантирует наличие общего интерфейса у каждого объекта в иерархии, и поэтому всякая сущность в составе этой иерархии, в конечном счете, принадлежит к одному базовому типу. Один из самых серьезных минусов C++ – отсутствие поддержки подобной иерархии. Рассмотрим простой пример.

Допустим, создав иерархию объектов на C++ на основе собственного базового класса CFoo, вы хотите интегрировать ее с другой иерархией, все объекты которой происходят от базового класса CBar. В этом примере интерфейсы иерархий объектов несовместимы, поэтому их интеграция заставит попотеть. Чтобы справиться с этой задачей, вам придется использовать класс-оболочку или множественное наследование. В случае иерархии с од-

ним корнем совместимость не проблема, так как интерфейс объектов един (унаследованный от System.Object). В результате вы знаете, что у любого и каждого объекта вашей иерархии – и, что важнее всего, в иерархиях кода .NET сторонних разработчиков – всегда будет минимальный набор функциональности.

Безопасность типов. В завершение упомянем такое полезное свойство CTS, как безопасность типов. Безопасность типов гарантирует, что типы являются именно тем, за что они себя выдают, и что над некоторым типом можно выполнить лишь подходящие действия. Безопасность типов предоставляет несколько преимуществ и возможностей, большинство из которых обеспечиваются иерархией объектов с единым корнем.

• Каждая ссылка на объект типизирована, как и объект, на который она ссылается. Система CTS гарантирует, что ссылка всегда указывает именно на то, на что она должна указывать.

• Поскольку CTS отслеживает каждый тип в системе, систему нельзя обмануть, выдав один тип за другой. Очевидно, это особенно важно для распределенных приложений, где приоритетом является защита.

• Каждый тип отвечает за определение доступности своих членов, задавая модификатор доступа. Это делается для каждого члена в отдельности. Может быть задан любой вид доступа (если член объявлен как **public**), доступ может быть ограничен кругом унаследованных классов (если член объявлен как **protected**) или вовсе запрещен (при объявлении члена как **private**). Можно также разрешить доступ к члену только другим типам в составе текущего компилируемого модуля, если объявить его как **internal**.

2.2.5.2. Метаданные и отражение

Как уже говорилось в п. 2.2.4, CLS-совместимые компиляторы создают из нашего исходного кода код MSIL, подлежащий компиляции (с помощью JIT-компиляторов) перед своим выполнением. Помимо перевода исходного кода в MSIL-последовательность, CLS-совместимые компиляторы выполняют и другую столь же важную задачу: внедрение метаданных в выходной EXE-файл.

Метаданные (metadata) – это данные, описывающие другие данные. В нашем контексте – это набор программных элементов EXE-файла, таких как типы и реализации методов. Эти метаданные похожи на библиотеки типов

(typelib), генерируемые компонентами Component Object Model (COM). Метаданные, порождаемые .NET-компилятором, как правило, не только намного выразительнее и полнее, чем библиотеки типов COM, к которым мы успели привыкнуть. Метаданные также всегда внедрены в EXE-файл. Благодаря этому исключены случайные потери метаданных приложения и рассогласование файлов.

Причина использования метаданных очевидна. Благодаря этой технологии CLR узнает, какие во время выполнения потребуются типы, и какие методы должны быть вызваны. Это дает среде возможность выполнить должную настройку для более эффективного выполнения приложения. Механизм запроса метаданных называется отражением (reflection). Библиотеки классов .NET Framework имеют целый набор методов отражения, позволяющих любому приложению (и не только CLR) запросить метаданные другого приложения.

Такие инструменты, как Visual Studio .NET, используют методы отражения для реализации средств подобных IntelliSense. При наличии IntelliSense вы, набирая имя метода, видите на экране всплывающий список с аргументами этого метода. Среда Visual Studio .NET дополняет это средство, показывая еще и все члены типа. Об API отражения см. § 5.3.

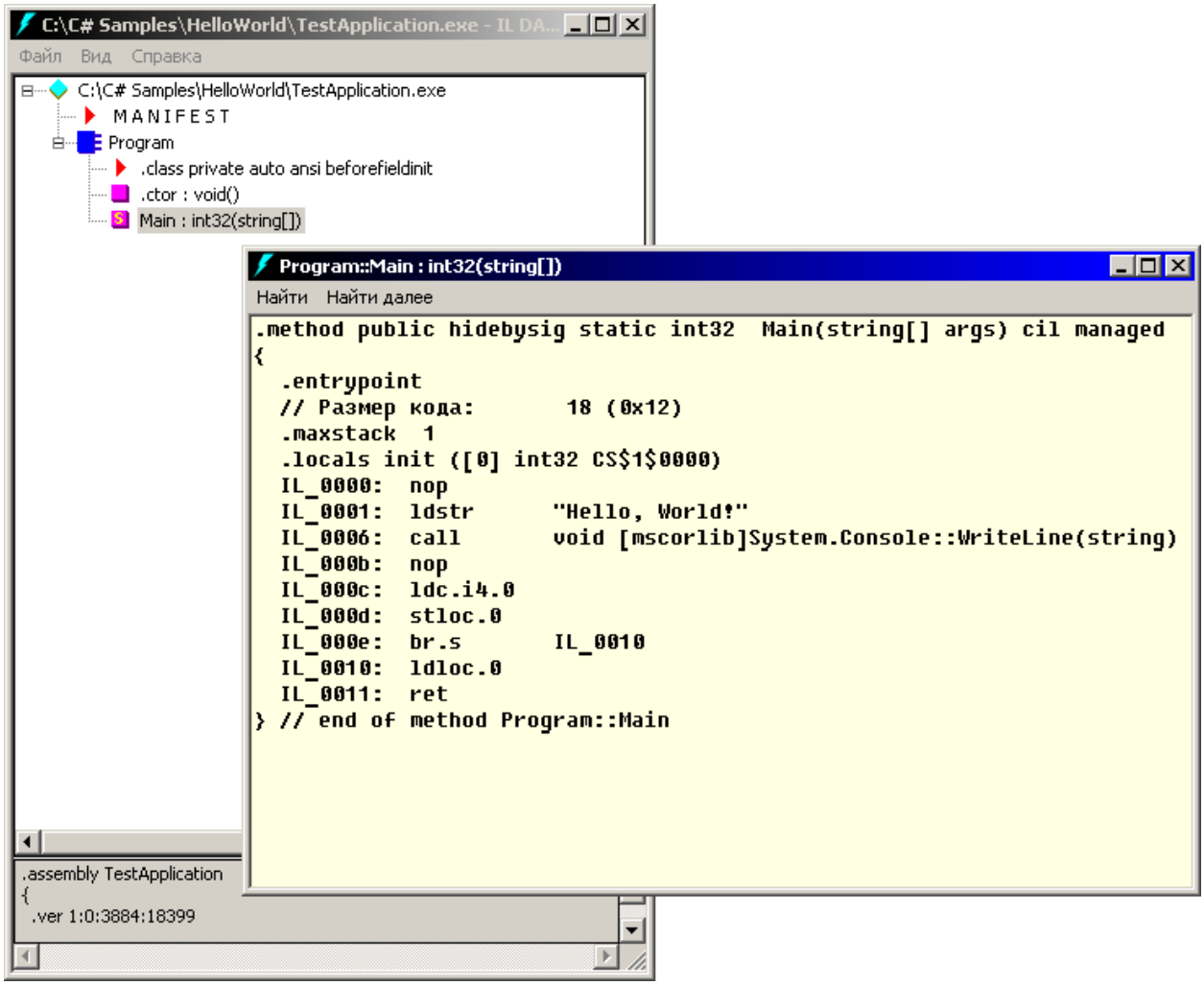


Рис. 2.2 – Приложение C# «Hello, World!», отображенное в ILDASM

Еще один чрезвычайно полезный инструмент .NET, использующий преимущество отражения – Microsoft .NET Framework IL Disassembler (ILDASM). Эта мощная утилита выполняет синтаксический разбор метаданных выбранного приложения, а затем отображает информацию о нем в виде дерева. На рисунке 2.2 показано, как выглядит приложение «Hello, World!» на C# в ILDASM.

На втором плане – главное окно IL Disassembler. Если дважды щелкнуть метод Main в иерархическом дереве, на переднем плане появится окно, отображающее подробности метода Main.

Как уже было сказано, компилятор .NET не создает исполняемые файлы (EXE и DLL) в традиционном виде. Вместо машинного кода он вставляет в них декларацию (manifest) со списками типов и классов, включенных в файл, а также коды MSIL-инструкций, подлежащих компиляции и выполнению либо установочным приложением, либо исполняющей средой .NET по-

средством JIT-компиляторов (JITter).

Очень важно, что сгенерированный MSIL-код похож на язык ассемблера и может использоваться в качестве учебного пособия, показывающего, что сделал компилятор с вашим кодом. Для просмотра выходного MSIL-файла можно использовать дисассемблер Microsoft .NET Framework IL Disassembler (ILDASM), позволяющий открыть выполняемый .NET-файл (EXE или DLL) и изучить его пространства имен, классы, типы и код (подробнее мы с ним познакомимся в п. 2.3.6).

2.2.6. Преимущества .NET

2.2.6.1. Безопасность

Самый важный аспект любой среды разработки распределенных приложений – способ обеспечения безопасности. Раньше бытовало мнение, что не стоит всерьез рассматривать Microsoft в отношении серверных решений для предприятий, пока она полностью не обновит подход к безопасности, в .NET появилось сразу несколько новых концепций. Работа системы безопасности начинается с того момента, когда CLR загружает класс, поскольку загрузчик классов является частью системы безопасности .NET. Так, при загрузке класса в .NET во время выполнения проверяются правила доступа и его внутренняя целостность. Кроме того, в ходе такой проверки выясняется, какая часть кода имеет надлежащие разрешения на доступ к определенным ресурсам. Система безопасности гарантирует проверку предписанных ролей и идентификационных данных. Чтобы не подвергать риску наиболее ответственные данные в распределенных вычислительных средах, эти проверки безопасности не ограничиваются рамками отдельных процессов и машин.

2.2.6.2. Развертывание

Развертывание – наиболее неприятная процедура разработки крупных распределенных систем. Любой разработчик серьезных Windows-программ может рассказать, как занимался при развертывании своих приложений размещением массы разнообразных двоичных и текстовых файлов, проблемами реестра Windows, компонентами COM, установкой библиотек поддержки таких продуктов, как Open Database Connectivity (ODBC) и Data Access Objects (DAO) и т.д. В противовес этому, развертывание – это та часть .NET, над ко-

торой проектировщики хорошо потрудились.

Ключ к развертыванию .NET-приложений – концепция сборок (assemblies). Сборкой называют пакет из семантически близких объектов, состоящий из одного или нескольких файлов. Особенности развертывания зависят от того, что вы разрабатываете – Web-серверное приложение или локальное приложение для Windows. Однако, с введением сборки как полностью инкапсулированного набора функциональных возможностей, развертывание сводится к простому копированию нужных сборок в место назначения.

Масса проблем, требующих усилий программистов до появления .NET Framework, теперь устранено. Теперь, например, не надо регистрировать компоненты (как это требуют COM и элементы управления ActiveX), поскольку благодаря метаданным и отражению все компоненты содержат в себе собственное описание. Во время выполнения .NET отслеживает также работу с файлами и версии файлов, связанных с приложением. Поэтому любое устанавливаемое приложение автоматически связывается с файлами, являющимися частью его сборки. Если программа установки попытается перезаписать файл, необходимый другому приложению, .NET разрешит установить нужные файлы, не удалив при этом предыдущие версии, поскольку они еще нужны другому приложению.

2.2.6.3. Взаимодействие с неуправляемым кодом

Неуправляемым (unmanaged code) называется код, который не находится под надзором .NET. Этот код тоже запускается средствами .NET, однако, у него нет тех преимуществ, которыми обладает управляемый код: сбора мусора, унифицированной системы типов и метаданных. Зачем тогда запускать неуправляемый код в среде .NET? Без особой нужды этого делать не стоит. Однако в некоторых случаях без этого не обойтись:

• Когда управляемый код вызывает функции неуправляемых библиотек DLL. Допустим, нашему приложению нужно работать с DLL, написанной на C, а компания, создавшая эту библиотеку, пока не адаптировала ее для технологии .NET. В этом случае нам придется по-прежнему вызывать эту DLL из .NET-приложения.

• Когда управляемый код использует компоненты COM. По той же причине, по какой нужно вызывать из своего приложения .NET функции DLL, написанной на C, нам требуется продолжать поддержку компонентов

COM. Выход в том, чтобы создать .NET-оболочку для компонента COM так, чтобы управляемый клиент полагал, будто он работает с .NET-классом.

• Когда пишется неуправляемый код, использующий .NET-службы. Здесь противоположная проблема: нам нужен доступ к .NET из неуправляемого кода. Она решается с помощью обратного подхода: клиент COM вводится в заблуждение, будто он работает с COM-сервером, который на самом деле является .NET-службой того же вида.

2.2.6.4. Очистка объектов и управление ресурсами

Возможность обеспечивать очистку и освобождение ресурсов после завершения исполнения компонентов – одна из самых важных функций системы на основе компонентов. Под «очисткой и освобождением ресурсов» понимается своевременное освобождение ссылок на другие компоненты и освобождение ресурсов, количество которых невелико или ограничено, за которые идет конкуренция (например, соединения с базой данных, описатели файлов и коммуникационные порты). Под «завершением» понимается тот момент, когда объект более не используется. В C++ очистку осуществляет деструктор (destructor) объекта – определенная для каждого объекта C++ функция, автоматически выполняемая при выходе объекта из области видимости. Однако, деструктор не вызывается автоматически для динамических объектов. В мире Microsoft .NET очистку объектов автоматически производит .NET Garbage Collector (GC). В силу некоторых причин эта стратегия противоречива, поскольку в отличие от предсказуемости C++, исполнение кода завершения объекта в решениях .NET основано на модели с отложенными вычислениями («lazy» model). Так, GC использует фоновые потоки, определяющие, что ссылок на объект больше не осталось. Другие потоки GC, в свою очередь, отвечают за исполнение кода завершения данного объекта.

§ 2.3. Hello, World!

В качестве примера создадим простейшую программу на языке C#, выводящую в консольное окно фразу «Hello, World!».

2.3.1. Выбор среды разработки

Прежде, чем приступать непосредственно к программированию, нужно определиться, с помощью чего мы будем создавать программы на языке C#.

Необходимо понимать, что среда разработки .NET и среда выполнения .NET Framework (.NET Framework Runtime) – независимые, в общем-то, друг от друга компоненты. Если на компьютере ранее уже были установлены программы, требующие наличия .NET Framework, то среда исполнения также может быть установлена вместе с ними. Это легко проверить – достаточно зайти в папку, где установлена операционная система Windows, и найти там папку Microsoft.NET\Framework (рис. 2.3).

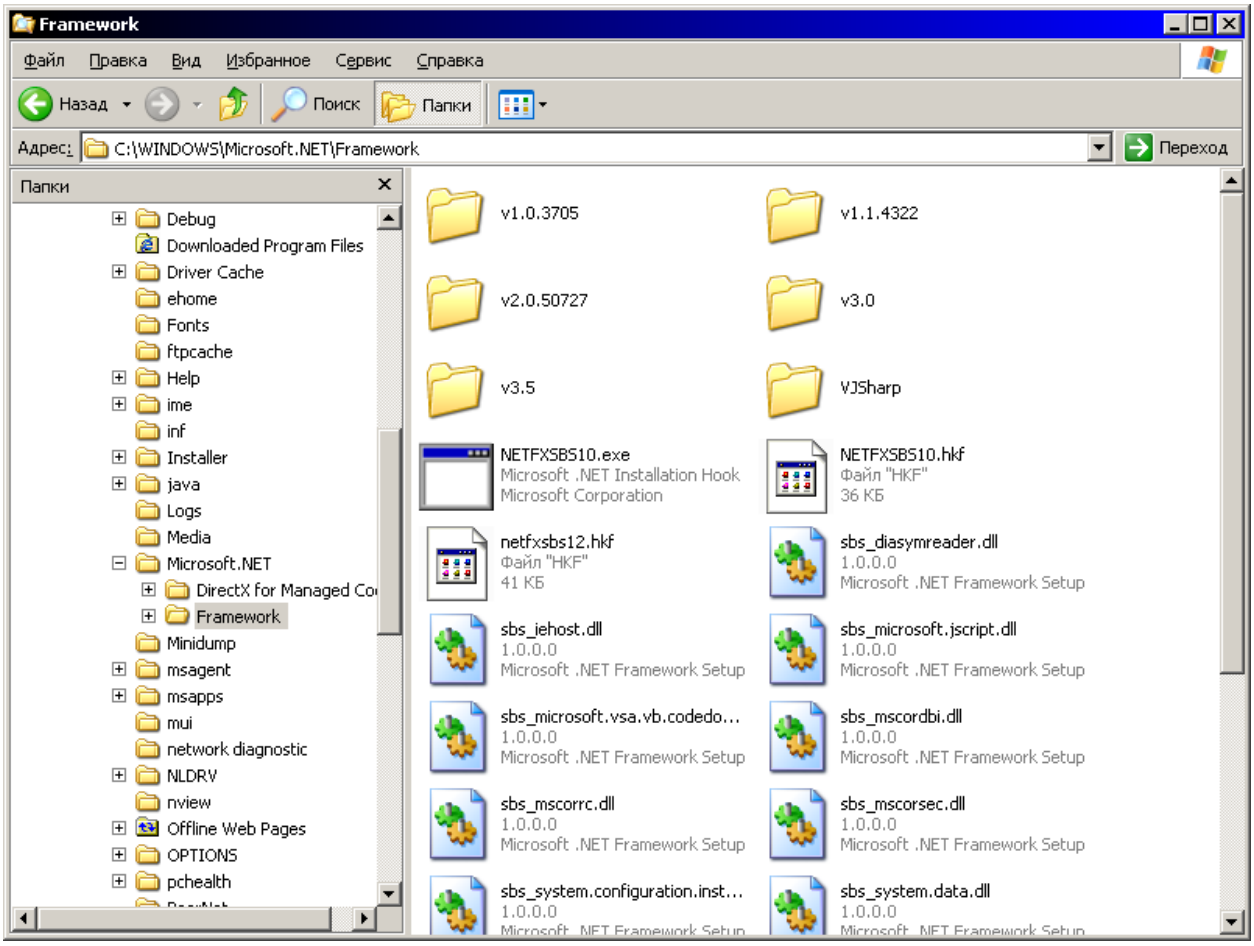


Рис. 2.3 – Установленные версии .NET Framework

Как видно из рисунка 2.3, на данном ПК установлены пять версий .NET Framework – 1.0, 1.1, 2.0, 3.0 и 3.5. В принципе, достаточно наличия одной версии, если она не ниже требуемой. Далее необходимо проверить, имеется ли для данной версии .NET Framework компилятор C#. Например, проверим версию 3.5 (рис. 2.4).

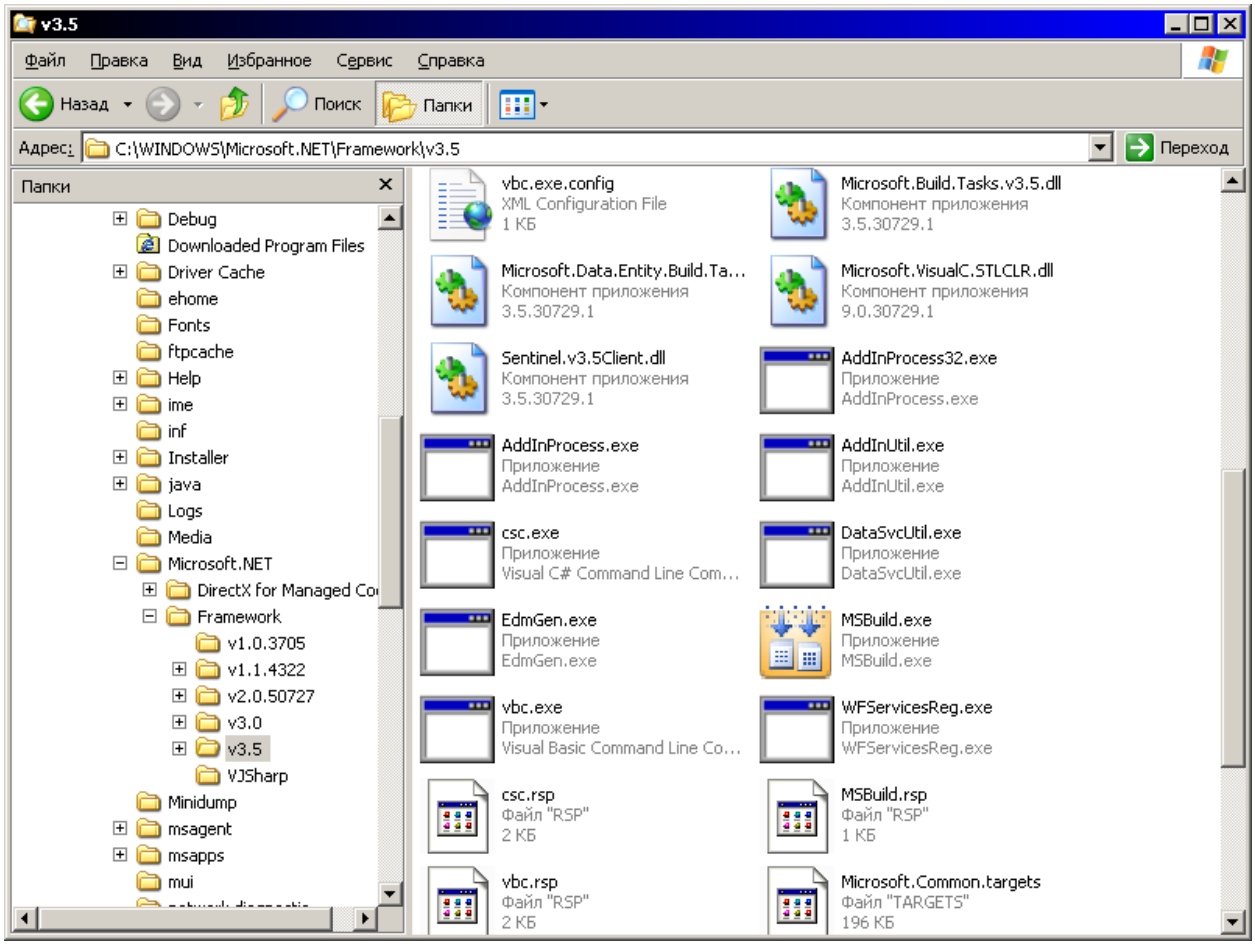


Рис. 2.4 – Установленные компоненты .NET Framework 3.5

Файл «CSC.EXE» – это и есть компилятор языка C#. Могут быть установлены и другие компиляторы:

- «JSC.EXE» – компилятор JScript;
- «VJC.EXE» – компилятор J#;
- «VBC.EXE» – компилятор Visual Basic;
- «CL.EXE» – компилятор Visual C++;

и т.д. Если компилятор C# отсутствует (либо отсутствует компилятор требуемой версии), то достаточно установить .NET Framework Runtime из центра загрузок Microsoft (microsoft.com/downloads/ru-ru). Размер полного дистрибутива версии 4.0 – около 50 Мб.

В качестве среды разработки можно использовать:

• обычный Блокнот Windows;

• текстовый редактор с подсветкой синтаксиса, например, Notepad++, который можно скачать с сайта разработчиков notepad-plus-plus.org/ru/downloads, с сайта sourceforge.net/projects/notepad-plus/files/notepad%2B%2B%20releases%20binary/ (размер дистрибутива версии 5.6 – 3 Мб, версии 5.7 – 4 Мб, к учебнику прилагается дистрибутив версии 5.6.7);

• специализированная среда разработки, например, SharpDevelop, которую можно скачать с сайта разработчиков www.icsharpcode.net/OpenSource/SD/Download/Default.aspx (размер дистрибутива версии 2.2 – 8 Мб, версии 3.2 – 15 Мб, к учебнику прилагается дистрибутив версии 3.2).

Однако, учитывая, что Microsoft, как разработчик технологии .NET Framework и языка C#, предоставляет собственную официальную бесплатную версию среды разработки Visual Studio Express Edition, можно скачать (с сайта Microsoft или любого другого) и установить ее, при этом будет также установлена и соответствующая версия .NET Framework Runtime. Бесплатная версия включает такие продукты, как Visual C#, Visual Basic, Visual C++ и Visual Web Developer. Полная версия дистрибутива Visual Studio 2008 Express Edition имеет объем около 820 Мб (включает .NET Framework Runtime 3.5), Visual Studio 2010 Express Edition – 1,7 Гб (включает .NET Framework Runtime 4.0).

Отдельно стоит остановиться на справочной системе – Microsoft Developer Network Library (библиотека MSDN). Имеется тенденция по переносу справочной системы в Internet (сайт msdn.microsoft.com/ru-ru/library). Такой подход имеет как плюсы (оперативное обновление справочной системы), так и минусы – невысокая скорость доступа к справочной системе (по сравнению с локальной справкой), постоянный внешний сетевой трафик. Причем, начиная с Visual Studio 2010, локальная версия справочной системы больше не выпускается. У пользователей Visual Studio 2008 есть выбор – можно использовать справку по сети, а можно установить локальную версию MSDN Library for Visual Studio 2008. Русифицированный вариант дистрибутива MSDN Library for Visual Studio 2008 имеет размер около 2,2 Гб. При этом можно выбирать, какой источник справки будет первичным (рис. 2.5).

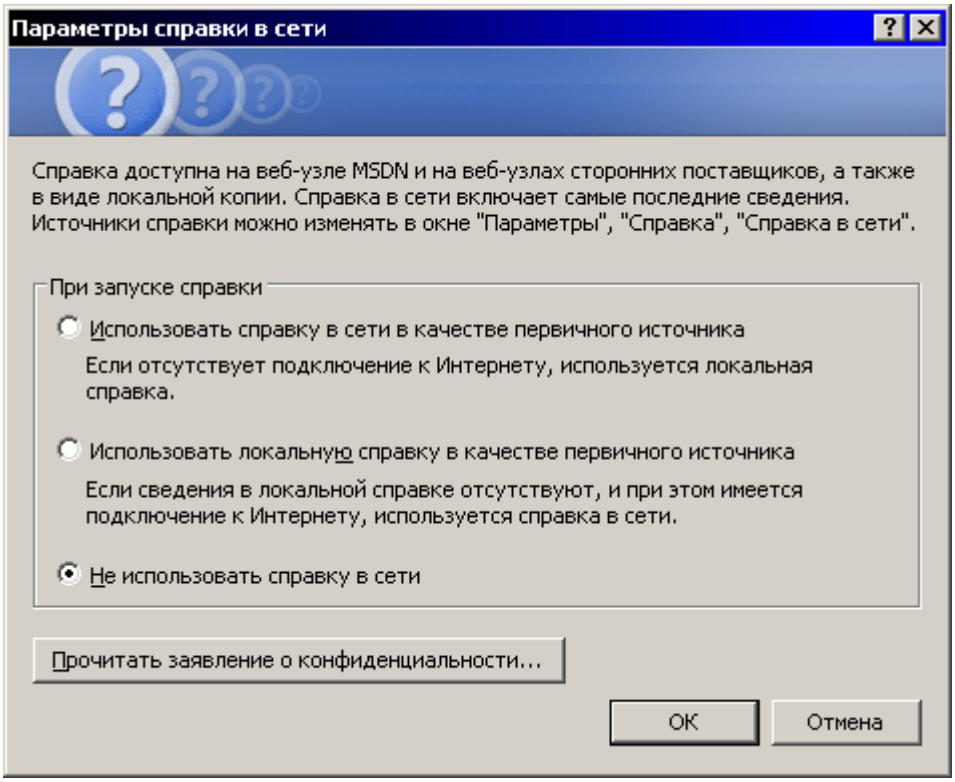


Рис. 2.5 – Выбор первичного источника справки

2.3.2. Создание программы

Рассмотрим все три варианта написания исходного кода программы – в обычном текстовом редакторе, в специализированной среде и в официальной среде разработки.

2.3.2.1. Использование текстового редактора

Запускаем Блокнот Windows (меню «Пуск → Программы → Стандартные → Блокнот» или «Пуск → Выполнить… → notepad») и набираем следующий текст (его анализ мы осуществим позже):

```
using System;

namespace Sample_2_3
{
	class Program
	{
		public static int Main(string[] args)
		{
			Console.WriteLine("Hello, World!");
			return 0;
		}
	}
}
```

Затем сохраним текст в каком-либо файле с расширением .cs. Например, в файле с именем «Sample_2_3_text.cs» (рис. 2.6).

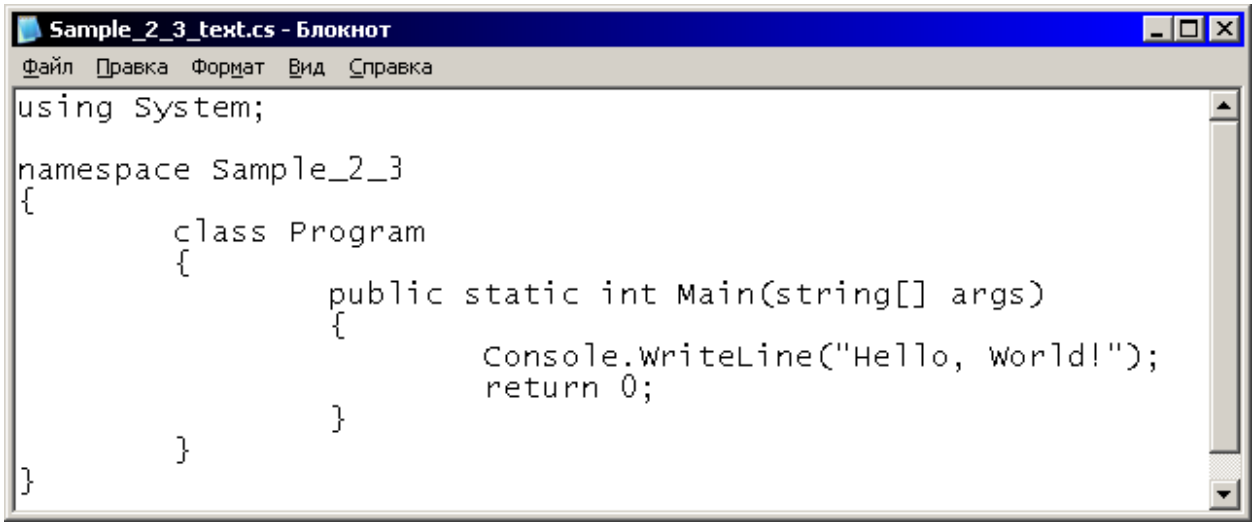


Рис. 2.6 – Программа на C# в Блокноте Windows

Можно использовать редактор с подсветкой синтаксиса. Например, редактор Notepad++ имеет варианты подсветки для многих языков, в т.ч. и для C# (рис. 2.7).

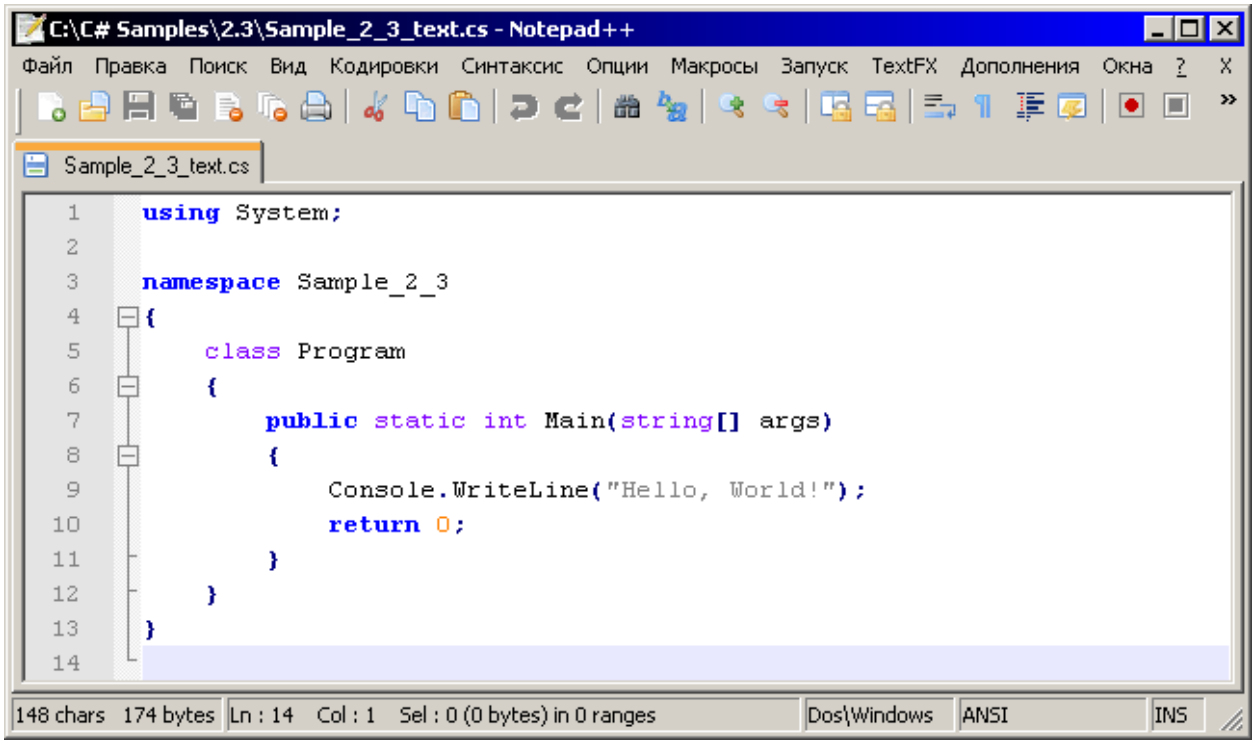


Рис. 2.7 – Программа на C# в редакторе Notepad++

Для набора текста программы можно также использовать файловые менеджеры со встроенными редакторами (Far Manager и т.д.). К ним также

можно найти расширения (плагины) для подсветки синтаксиса.

2.3.2.2. Использование специализированной среды

Устанавливаем среду SharpDevelop и запускаем ее. На стартовой странице (рис. 2.8) можно создать новое решение, нажав кнопку «New solution», либо использовав пункт меню «File → New → Solution…».

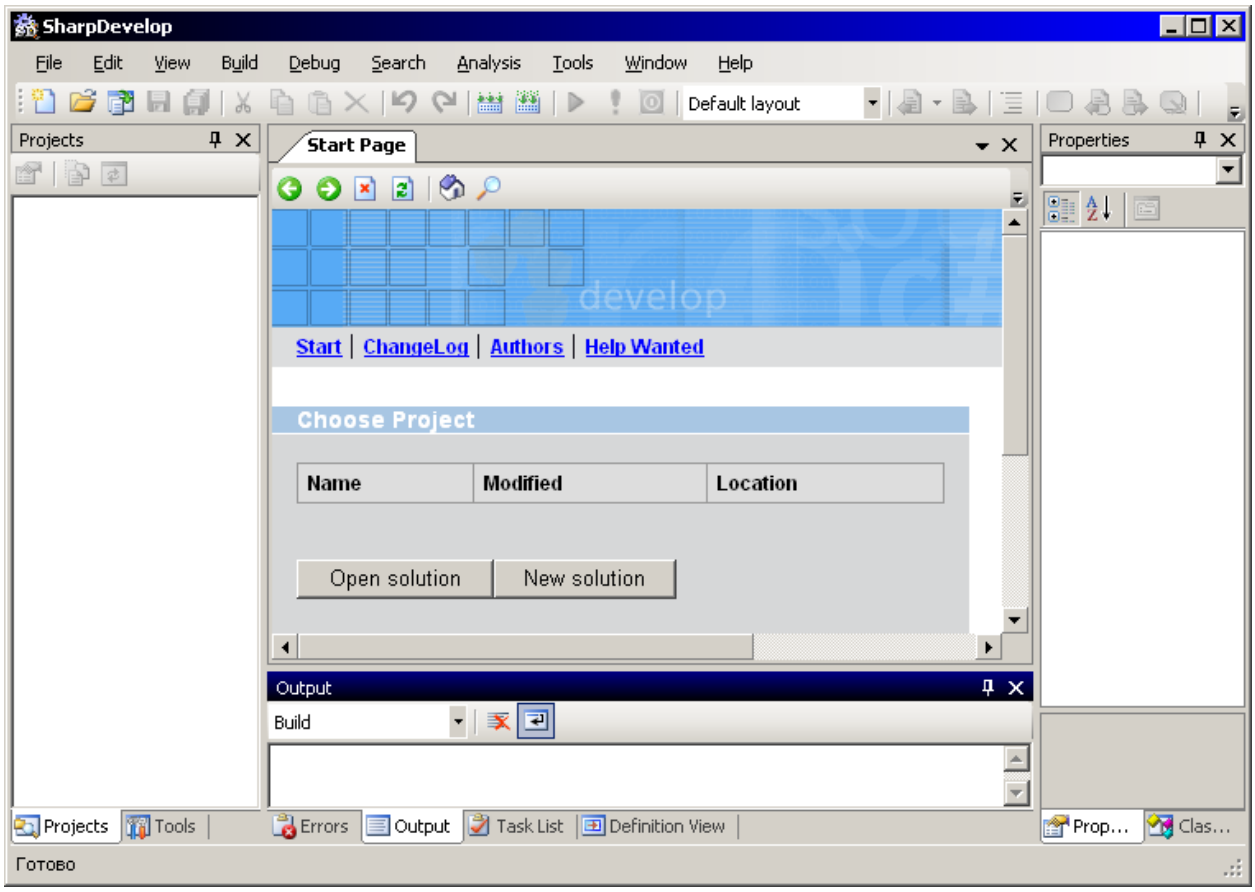


Рис. 2.8 – Стартовая страница SharpDevelop

После этого появляется диалог «New Project» (рис. 2.9). Следует выбрать тип проекта (в нашем случае это консольное приложение Windows на языке C#, поэтому выбираем категорию «C# → Windows Applications → Console Application»). Так как решение может состоять из нескольких проектов, есть отдельные поля для ввода имени проекта (Name) и имени решения (Solution name). Чтобы файлы разных проектов не перемешивались, для каждого проекта создается отдельная папка, имя которой совпадает с именем проекта. Для решения тоже можно создать отдельную папку, в которую будут помещаться папки всех проектов, входящих в решение. Но, т.к. наша программа имеет простую структуру, и в создаваемом решении будет всего один проект,

можно не создавать отдельную папку для решения. Для этого необходимо снять пометку «Create directory for solution». Введем текст «Sample_2_3_spec» в качестве имени проекта и решения (в общем случае, их имена не обязаны совпадать), этом случае проект будет помещен в папку «Sample_2_3_spec». Если бы мы не убрали отметку о создании отдельной папки для решения, проект был бы помещен в папку «Sample_2_3_spec\ Sample_2_3_spec».

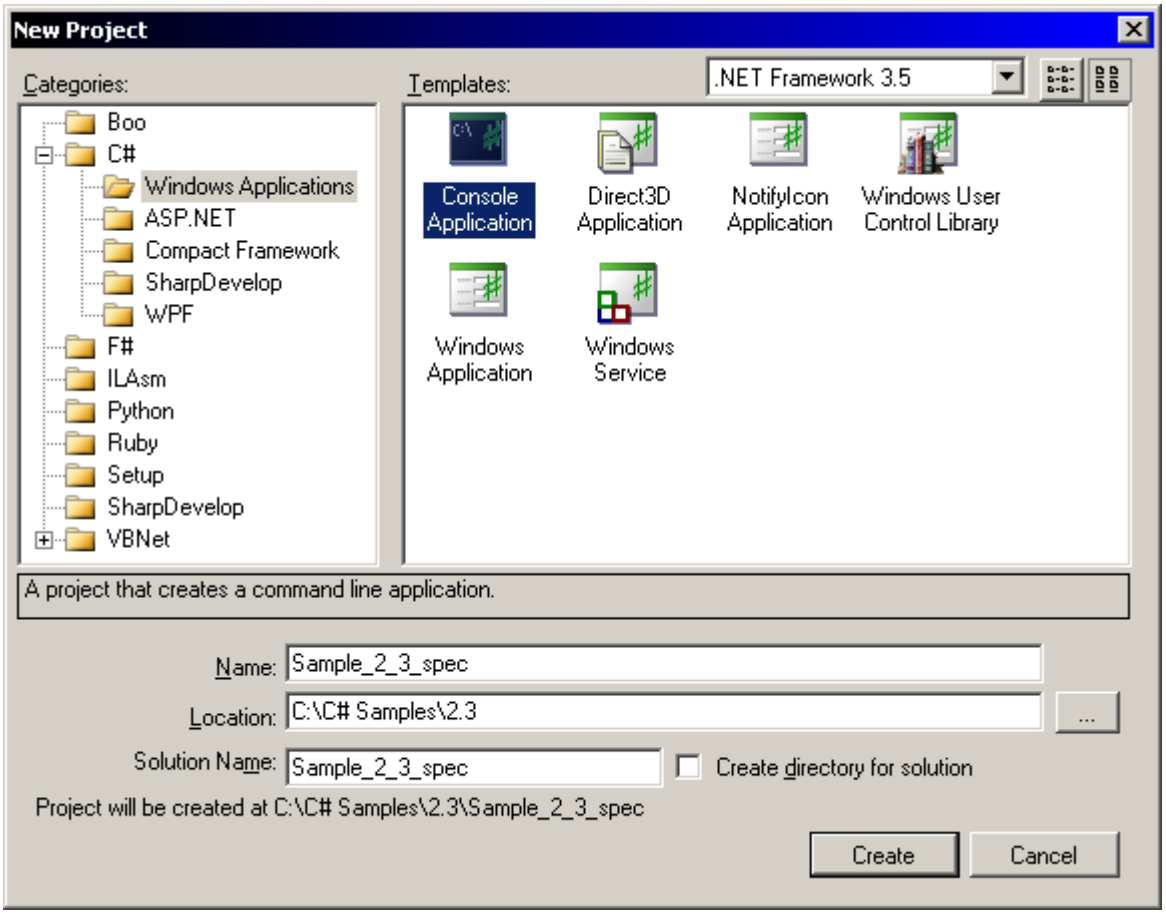


Рис. 2.9 – Создание проекта в среде SharpDevelop

Далее нажимаем кнопку «Create» и получаем проект, состоящий из нескольких файлов. Их назначение мы рассмотрим позже, а пока нас интересует исходных файл «Program.cs». Он уже содержит некоторый код, но мы его удалим, оставив только тот текст, что был приведен выше (рис. 2.10). Можно его скопировать из созданного ранее текстового файла, а можно набрать заново, используя автоматическое завершение ввода (о нем ниже) и автоматическое форматирование программы (среда сама расставляет отступы блоков, закрывающие фигурные скобки и т.д.).

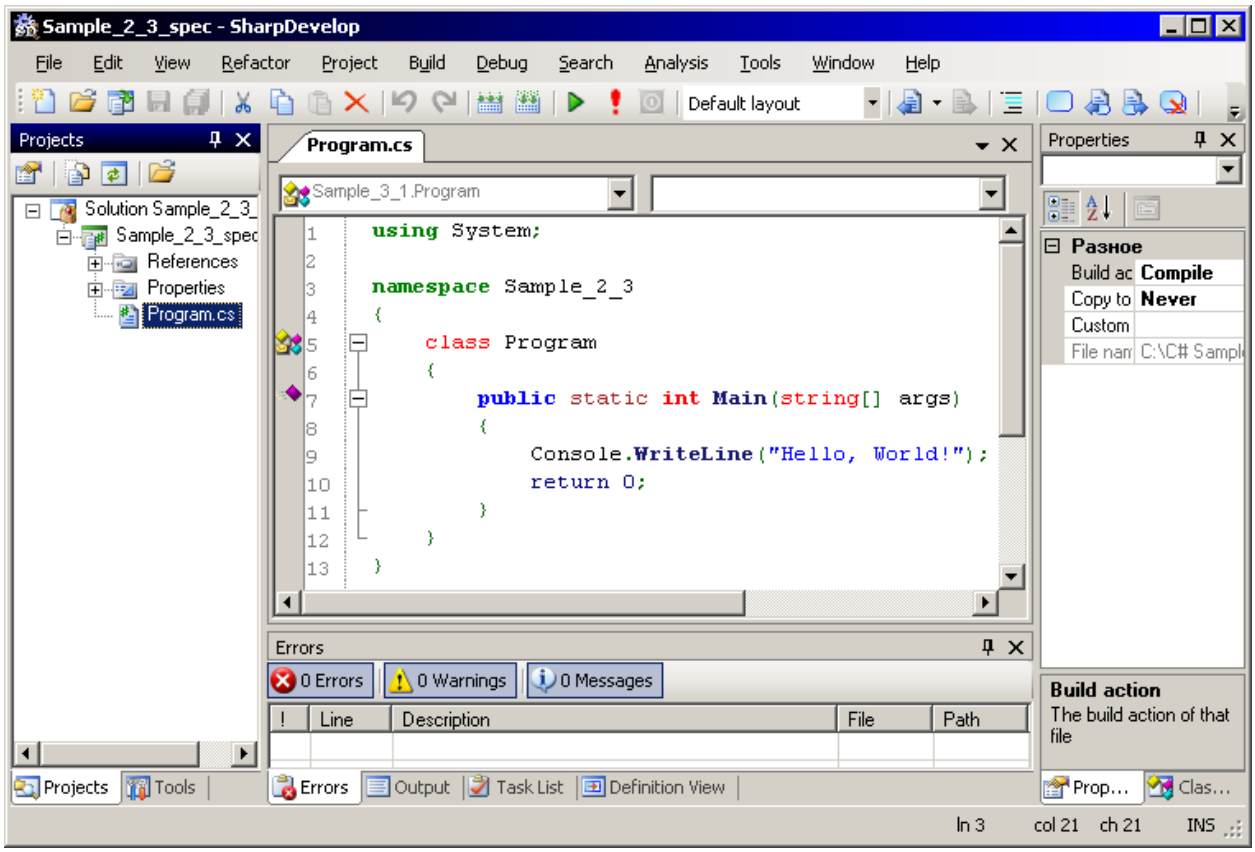


Рис. 2.10 – Программа на C# в среде SharpDevelop

После этого сохраним проект, используя пункт меню «File → Save All» или соответствующую кнопку на панели инструментов (горячая клавиша – Ctrl+Shift+S).

Видно, что по сравнению с текстовым редактором, даже обладающим подсветкой синтаксиса, редактор кода среды SharpDevelop обладает дополнительными возможностями:

• Специальными значками на полях редактора кода помечаются классы и члены классов.

• Поддерживается автозавершение ввода. Т.е., когда программист набирает код, программа предлагает ему в выпадающем списке варианты его завершения (либо можно вызвать этот список, нажав комбинацию клавиш Ctrl+Пробел). Это существенно ускоряет набор текста программы.

• Используется древовидная структура программы. Поэтому существует возможность «свернуть» часть кода в одну строку, нажав на значке «⊟». Это удобно при разработке больших программ – развернутым остается только редактируемый и отлаживаемый код.

2.3.2.3. Использование официальной среды разработки

В пакете Microsoft Visual Studio 2008 Express Edition среда разработки для каждого языка программирования загружается отдельно. Так, на рисунке 2.11 показана начальная страница среды разработки для Visual C#. Для создания проекта необходимо выбрать пункт меню «Файл → Создать проект…» или нажать на ссылку «Проект…» напротив пункта «Создать» на стартовой странице.

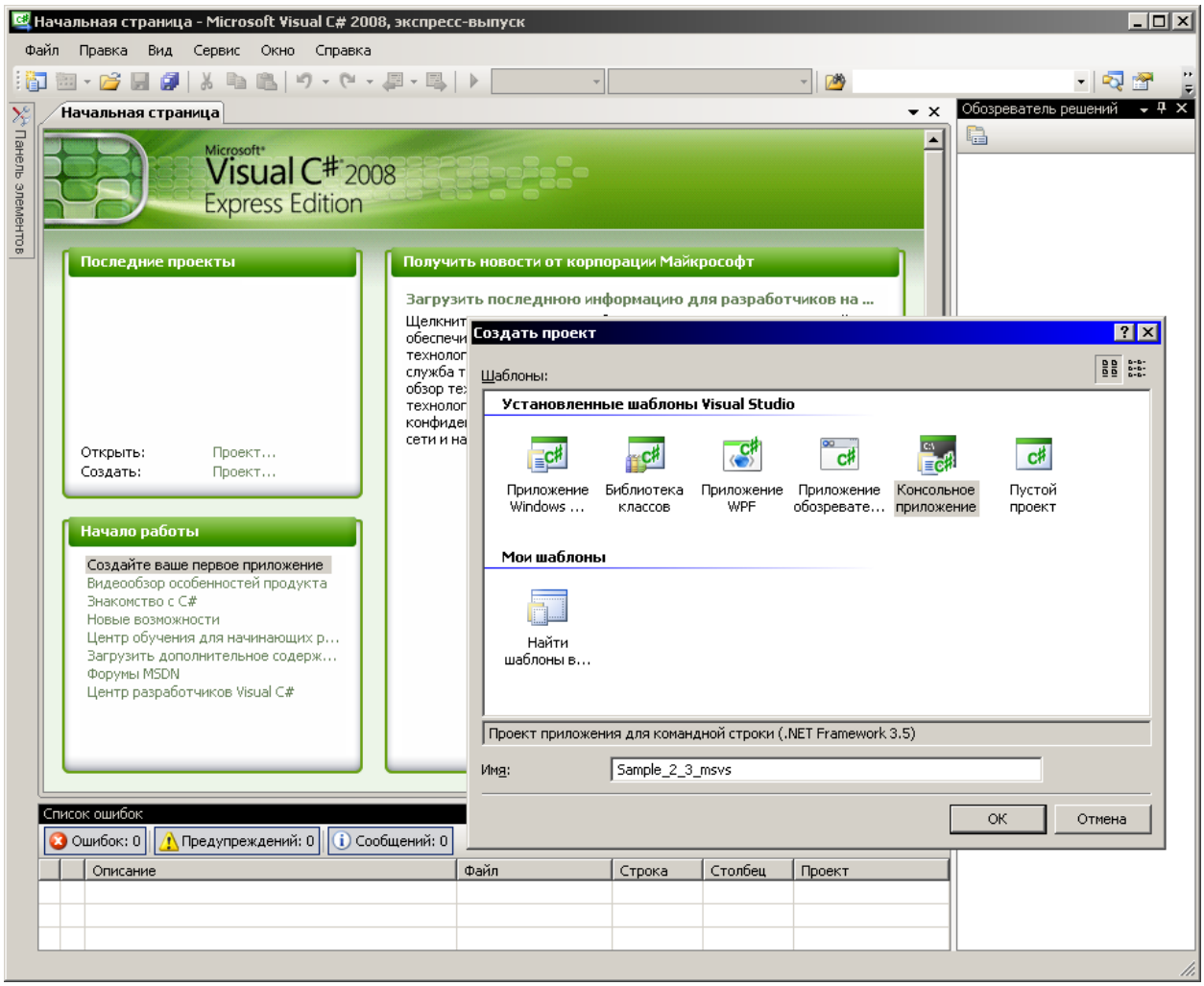


Рис. 2.11 – Создание проекта в среде Visual Studio 2008 Express Edition

Видно, что возможности создания проекта в Express-версии существенно ограничены даже по сравнению со средой SharpDevelop:

• поддерживается меньше типов проектов;

• для каждого языка предназначена своя среда разработки, следовательно, нельзя тестировать проект, исходные файлы в котором написаны на разных языках программирования;

• нет возможности указать имя решения и папки для создания проекта (по умолчанию используется «Мои документы\Visual Studio 2008\Projects») и т.д.

Имя решения и папки можно указать позже, при первом сохранении проекта (рис. 2.12). Назовем наш проект «Sample_2_3_msvs».

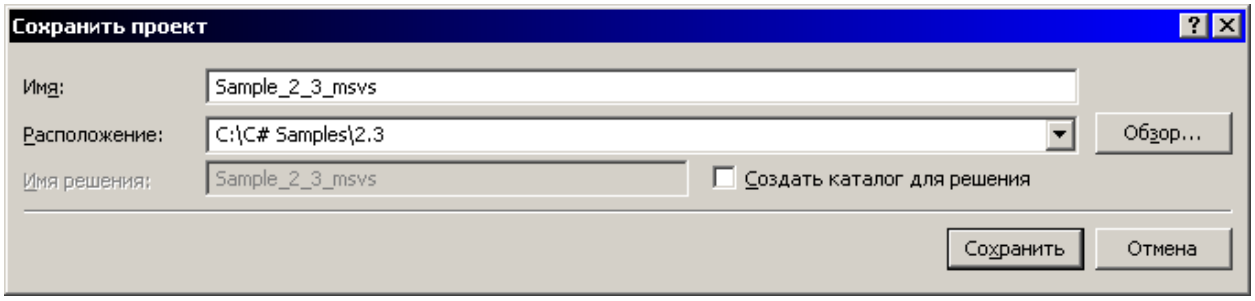


Рис. 2.12 – Сохранение проекта в среде Visual Studio 2008 Express Edition

В полной платной версии Visual Studio, например, Visual Studio 2008 Professional (рис. 2.13) изначально есть те опции по созданию проекта, которые мы видели в среде SharpDevelop.

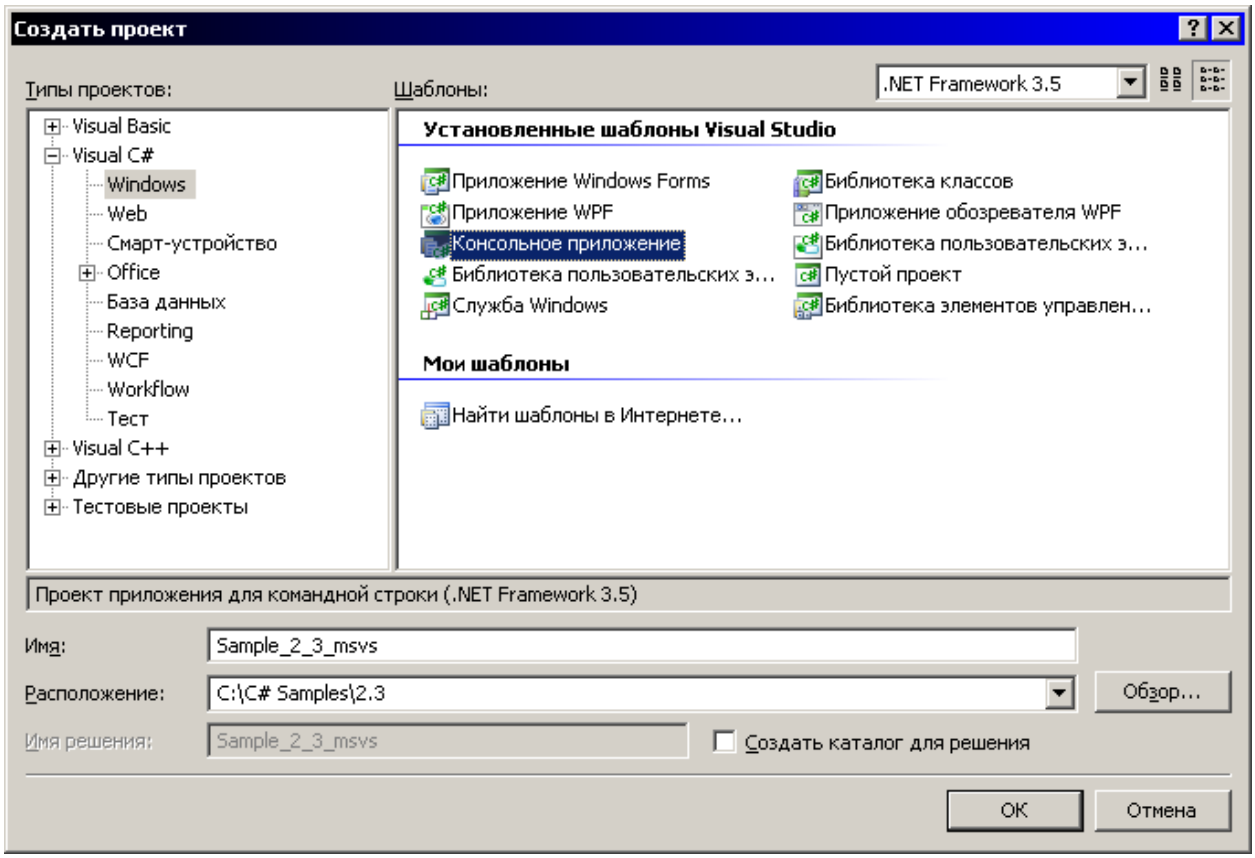


Рис. 2.13 – Создание проекта в среде Visual Studio 2008 Professional

Ну а после взгляда на редактор кода Microsoft Visual Studio оконча-

тельно становится понятно, что среда SharpDevelop разрабатывалась с оглядкой на Visual Studio. Внешний вид многих окон и основные возможности в них совпадают (рис. 2.14). Наберем тот же текст программы (автоматическое форматирование в Visual Studio несколько отличается от того, как оно реализовано в SharpDevelop) или скопируем его.

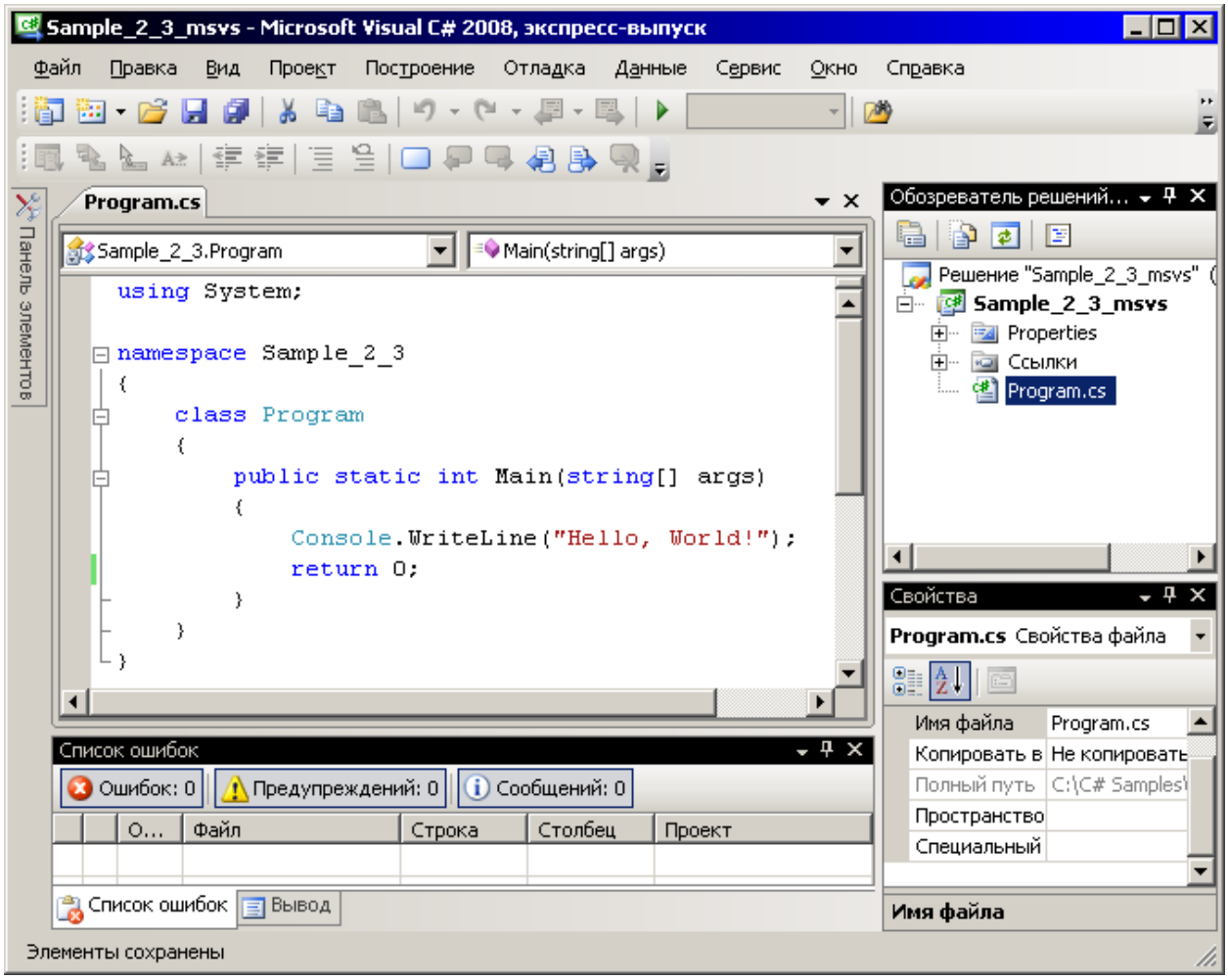


Рис. 2.14 – Программа на C# в среде Visual Studio 2008 Express Edition

2.3.3. Компиляция

Итак, мы получили три версии первой программы, пора переходить к компиляции.

2.3.3.1. Компиляция текстового файла

Выше уже было показано (рис. 2.4), что компилятор языка C# – это файл со спецификацией

```
<папка Windows>\Microsoft.NET\Framework\<номер версии>\csc.exe
```

Обычно это соответствует

```
C:\WINDOWS\Microsoft.NET\Framework\v3.5\csc.exe
```

В качестве параметра компилятору указываем имя исходного файла (рис. 2.15).

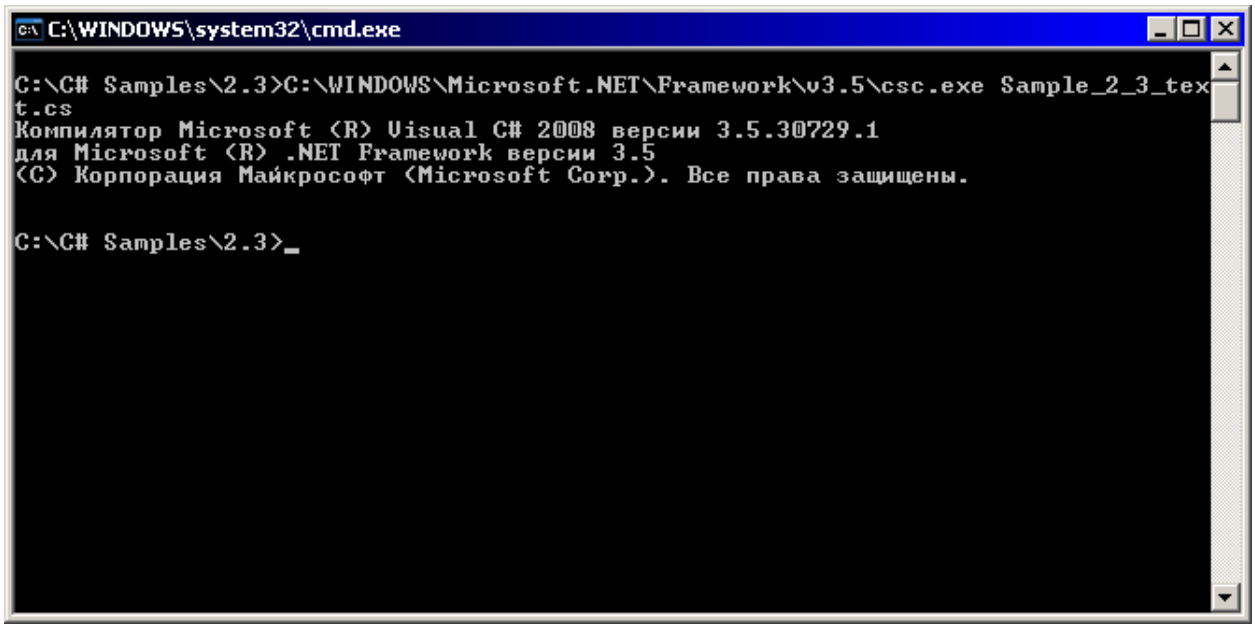


Рис. 2.15 – Компиляция программы на языке C# из консоли

Если при компиляции возникнут ошибки, они будут отражены на консоли. В противном случае в папке с программой появится исполняемый файл с таким же именем, как у исходной программы, и расширением .EXE.

Чтобы каждый раз не набирать данную строку в консоли, можно в папке с исходным файлом сделать пакетный файл (например, с именем CS23.BAT) следующего содержимого:

```
C:\WINDOWS\Microsoft.NET\Framework\v3.5\csc.exe Sample_2_3_text.cs
```

Такой файл можно выполнять не только на консоли, но и из папки Windows (рис. 2.16). Если с помощью пакетного файла планируется компиляция различных программ, следует использовать следующую запись:

```
C:\WINDOWS\Microsoft.NET\Framework\v3.5\csc.exe %1
```

В этом случае будет компилироваться файл с именем, указанным в качестве первого параметра пакетного файла. Назовем этот файл CS.BAT, тогда использовать его в консоли можно так:

```
cs.bat Sample_2_3_text.cs
```

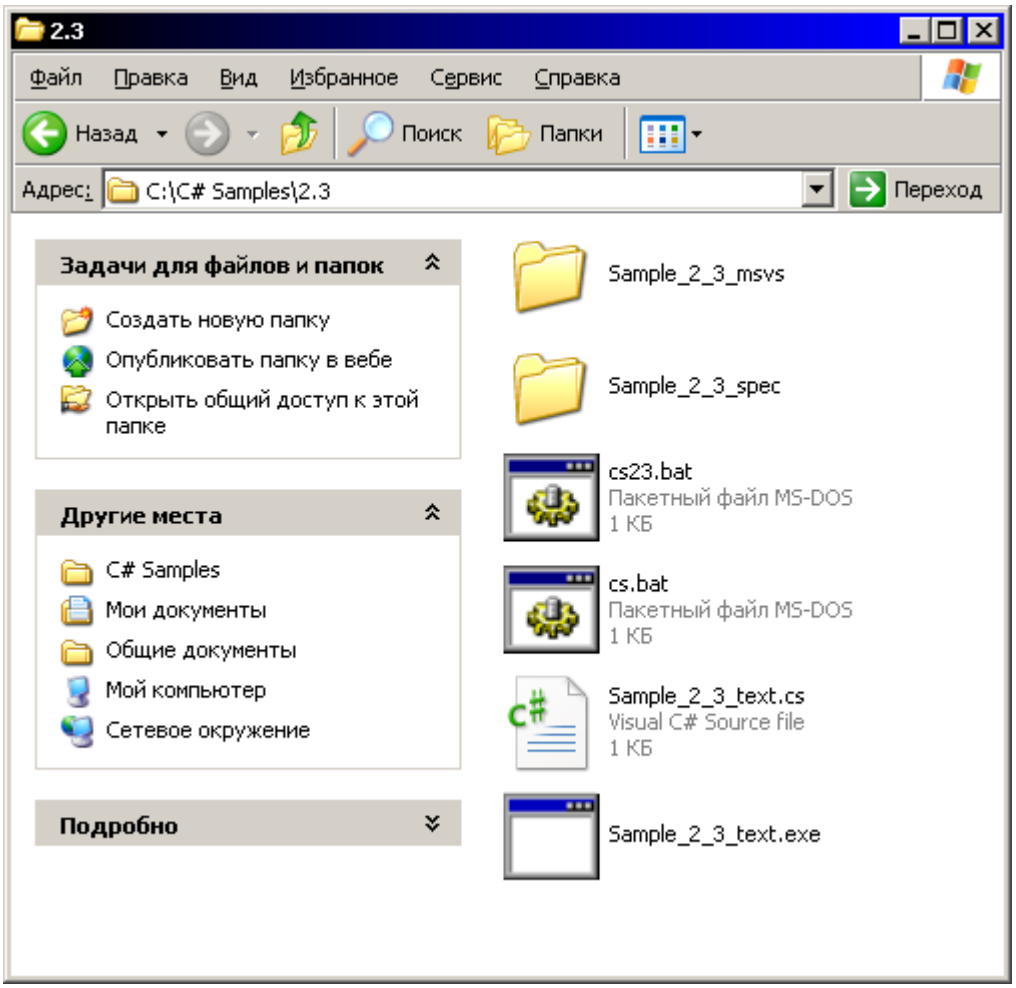


Рис. 2.16 – Файловая структура примеров к § 2.3

С таким файлом удобнее всего работать из файлового менеджера. Например, Far Manager является консольным, и при нажатии комбинации клавиш Ctrl+Enter копирует имя файла под курсором в командную строку.

2.3.3.2. Компиляция проекта в среде SharpDevelop

В меню «Build» есть несколько пунктов для компиляции:

1. «Build Solution» (F8) – компилирует решение;
2. «Rebuild Solution» (Alt+F8) – перекомпилирует решение;
3. «Build Project» (F9) – компилирует проект;
4. «Rebuild Project» (Alt+F9) – перекомпилирует проект.

Чем отличается компиляция от перекомпиляции? При компиляции, если решение уже компилировалось ранее, будут скомпилированы только те файлы проекта или решения, которые с тех пор подверглись модификации. При перекомпиляции будут заново скомпилированы все файлы проекта или

решения. Перекомпиляция может потребоваться в следующих случаях:

• Какой-либо файл был модифицирован в других проектах, решениях или внешних редакторах, а среда Visual Studio об этом «не узнала»;

• Были изменены настройки среды или операционной системы, влияющие на результат компиляции.

В пунктах 3 и 4 вместо слова «Project» подставляется актуальное имя проекта. Так как в нашем решении всего один проект, то результаты компиляции проекта и решения совпадают.

Исполняемый файл появляется в подпапке «bin\Debug» папки проекта, если компилировалась отладочная версия, и в подпапке «bin\Release», если компилировалась финальная версия (релиз).

2.3.3.3. Компиляция проекта в среде Visual Studio

В целом, совпадает с компиляцией в среде SharpDevelop. Естественно, версия Express Edition обладает меньшим набором возможностей, в меню «Построение» есть только пункт «Построить решение» (F6) и «Перестроить решение».

В полной версии есть все четыре пункта – «Построить решение» (Ctrl+Shift+B), «Перестроить решение», «Построить проект», «Перестроить проект». Вместо слова «проект» подставляется актуальное имя проекта.

2.3.4. Выполнение

Для выполнения программы извне среды разработки достаточно запустить исполняемый файл, полученный на этапе компиляции. Сделать это можно из консоли (рис. 2.17) или из папки программы, открытой в Проводнике.

При запуске программы из среды разработки можно использовать следующие команды (меню «Debug» в среде SharpDevelop / «Отладка» в среде Visual Studio):

• «Run» / «Начать отладку» (F5) – запуск программы;

• «Run without debugger» / «Запуск без отладки» (Ctrl+F5) – запуск программы без отладки. При этом будут игнорироваться все точки останова (о которых мы будем говорить в 5-й главе) и прочие средства отладки.

Однако, что же происходит при выполнении программы? При запуске программы из Проводника, из среды SharpDevelop, а также при запуске с от-

ладкой из среды Visual Studio мы не имеем возможности наблюдения за результатами ее работы. Окно с результатами работы программы появляется на короткое время, и тут же исчезает.

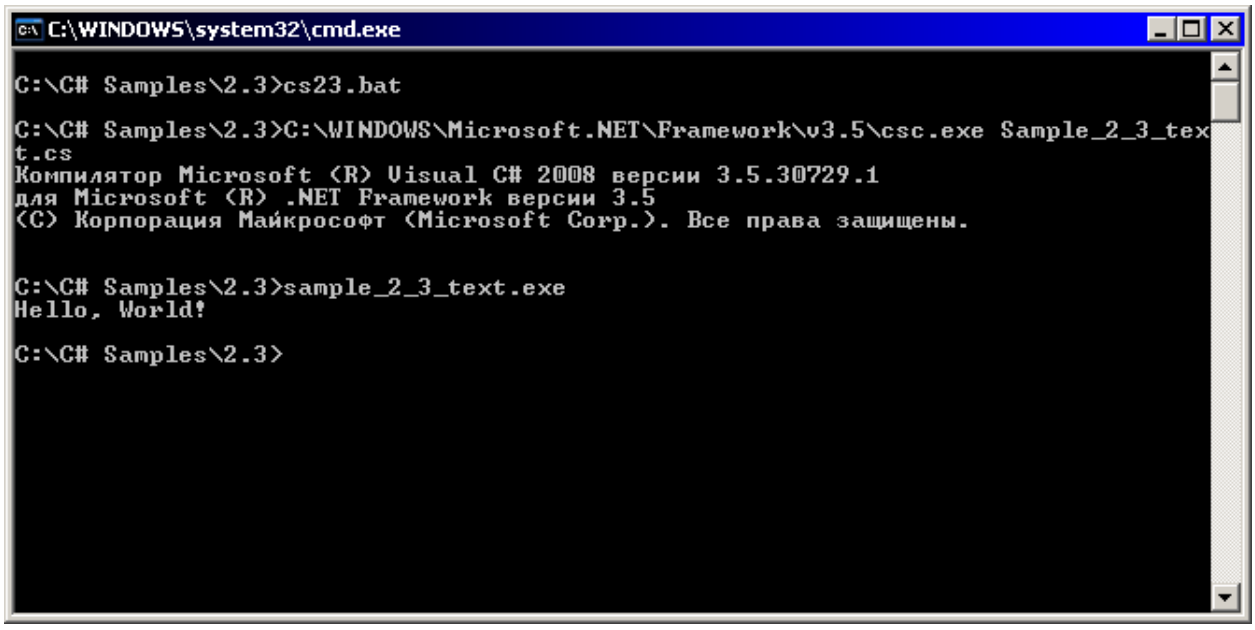


Рис. 2.17 – Выполнение программы из консоли

И только при запуске из среды Visual Studio без отладки мы видим результат (рис. 2.18).

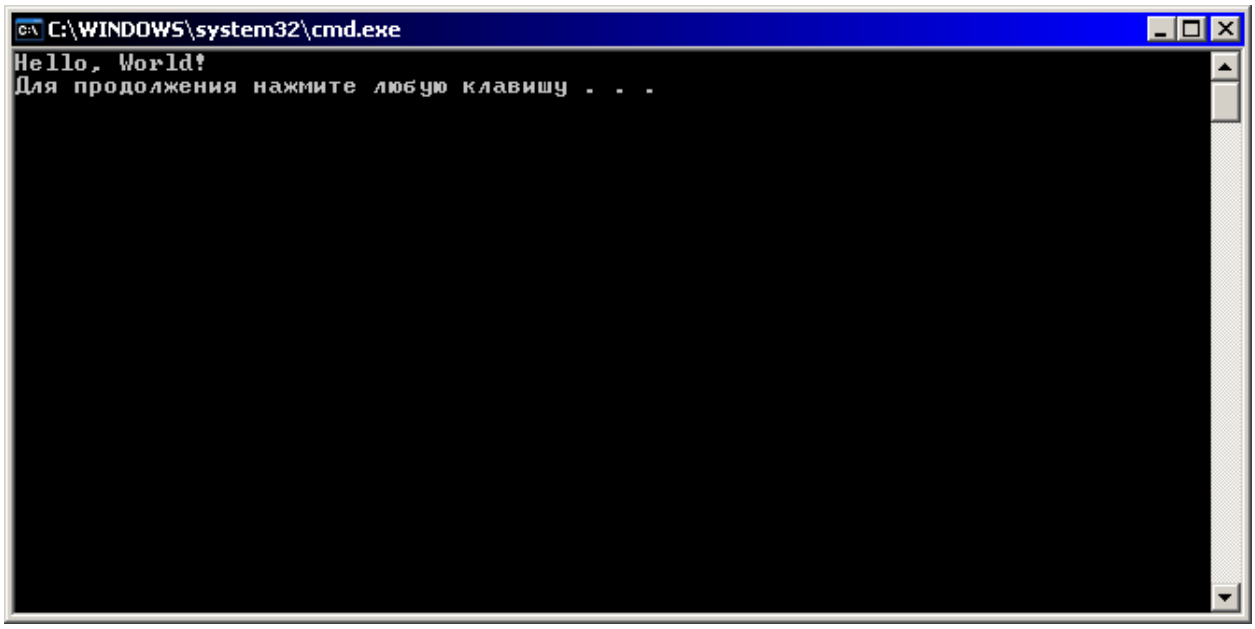


Рис. 2.18 – Запуск программы без отладки в среде Visual Studio

Все дело в том, что с выводом на консоль строки «Hello, World!» программа завершает свою работу, возвращая операционной системе код возвра-

та (0). Запуск без отладки в среде Visual Studio настроен таким образом, что после завершения программы ожидается нажатие любой клавиши, в остальных же случаях консольное окно сразу закрывается. Чтобы этого не происходило, код можно несколько модифицировать:

```
using System;

namespace Sample_2_3
{
	class Program
	{
		public static int Main(string[] args)
		{
			Console.WriteLine("Hello, World!");
			Console.ReadKey(true);
			return 0;
		}
	}
}
```

В этом случае программа будет ожидать нажатия любой клавиши, и только после этого завершит свою работу.

2.3.5. Анализ исходного кода

Проанализируем набранный нами код, сравнив его с программой на языке C++, выполняющей аналогичные функции:

```
#include <stdio.h> // или <iostream>
#include <conio.h> // или <cconio>

int main(int argc, char *args[])
{
	printf("Hello, World!"); // или cout << "Hello, World!"
	getch();
	return 0;
}
```

1) **using** System

В языке C++, чтобы получить доступ к библиотечным функциям, мы подключали к программе соответствующие библиотеки (директивой препроцессора #include). В языке C# нет аналогичного языку C++ понятия библиотек, подключаются только пространства имен. При этом можно использовать все объекты, содержащиеся в подключенном пространстве имен, и не важно, в каком именно файле они описаны (объекты пространства имен могут быть размещены в нескольких различных файлах). В данном случае мы подключаем пространство имен System, которое содержит класс Console, обеспечивающий консольный ввод и вывод, а также классы типов данных (**int**, **string** и

т.д.).

2) **namespace** Sample_2_3

Далее в программе на языке C# мы описывали пространство имен Sample_2_3. В принципе, делать это не обязательно – если объект разместить вне пространства имен, он автоматически будет включен в пространство имен по умолчанию. Но считается хорошим тоном не поступать таким образом. Включение объекта в пространство имен упрощает его использование из других модулей в дальнейшем. В языке C++ тоже можно использовать пространства имен, но т.к. в основе модулей языка лежали библиотеки, пространства имен использовались реже. Речь не идет о языке Visual C++ .NET, который подвергся некоторой модификации.

3) **class** Program

В отличие от языка C++, все объекты языка C# являются классами. Соответственно, нельзя описать процедуру или функцию вне класса, все процедуры и функции должны быть описаны как методы класса, в т.ч. и функция main. Название у данного класса может быть любым.

4) **public static int** Main(**string**[] args)

Т.к. Main – это метод класса, то для него должен быть указан модификатор доступа. В данном случае это **public**. Как и в языке C++, по умолчанию используется модификатор **private**. Модификатор **static** играет ту же роль, что и в языке C++, т.е. указывает, что метод является статическим и может быть вызван без создания экземпляра класса. Это обязательное требование для функции Main, т.к. компилятор автоматически не создает экземпляр класса, в котором эта функция описана (в нашем случае – Program). Имя функции, в отличие от языка C++, пишется с заглавной буквы. Возвращает функция, как и в языке C++, целое число – код возврата операционной системе. Аргументы функции Main играют ту же роль, это список параметров командной строки. Но, если в языке C++ было два аргумента (количество аргументов и массив указателей на них), то в языке C# аргумент один. Почему – станет понятно после изучения свойств массивов. Если анализ аргументов командной строки в программе не предусмотрен, скобки можно оставить пустыми:

```
    public static int Main()
```

Также следует отметить, что в языке C++ описание метода может находиться в классе, а реализация – как здесь же в классе (тогда компилятор не-

явно подразумевает модификатор inline), так и вне класса. Чаще всего описания находятся в заголовочных файлах (.H), а реализация – в исходных (.CPP). В языке C# выбора нет – реализация метода должна находиться в классе сразу после его описания, а понятие заголовочного файла отсутствует.

5) Console.WriteLine("Hello, World!")

Т.к. основу языка C# составляют классы, то и процедура вывода на консоль является методом класса (класса Console). Этот метод описан с модификаторами **public static**, поэтому доступен извне класса, и экземпляр класса Console для его вызова создавать не обязательно. Форматирование вывода в данном классе отличается от форматирования вывода функцией printf и классом ostream (экземпляром которого является cout), и мы его подробно рассмотрим далее. В отличие от языка C++, статические функции вызываются с использованием оператора «.» (класс.метод), в то время как в C++ предусмотрено два варианта: с использованием оператора «.» (экземпляр.метод) и оператора «::» (класс::метод).

6) Console.ReadKey(**true**)

Еще один статический метод класса Console, ожидает нажатия клавиши. Единственный параметр типа **bool** указывает, скрывать нажатую клавишу или вывести ее на консоль. Стоит отметить, что при доступе к классам пространство имен указывать не обязательно, хотя можно это сделать:

```
System.Console.ReadKey(true);
```

Пространство имен обязательно указывается только в том случае, если присутствует неоднозначность – несколько подключенных пространств имен содержат объект (класс, структуру и т.д.) с одинаковым именем.

7) **return** 0

Смысл этого оператора тот же, что и в языке C++. Возвращаем операционной системе код успешного завершения программы. Опять же, как и в языке C++ этот код можно не возвращать, тогда в описании функции Main меняем **int** на **void**:

```
public static void Main()
{
      Console.WriteLine("Hello, World!");
      Console.ReadKey(true);
}
```

2.3.6. Анализ кода MSIL

Итак, посмотрим, какой же код MSIL генерируется при компиляции нашего приложения. Для этого запустим Microsoft .NET Framework IL Disassembler (ILDASM). Запустить его можно из среды SharpDevelop, выбрав пункт меню «Tool» → «IL Dasm».

Если ссылка на ILDASM отсутствует в меню среды разработки, то можно, во-первых, запустить эту программу самостоятельно:

• Зайти в папку

```
<папка Program Files>\Microsoft SDKs\Windows\<номер версии SDK>\bin
```

(см. рис. 2.19) и запустить исполняемый файл ildasm.exe. В этой же папке располагается справка по этой программе – DASMHLP.HLP.

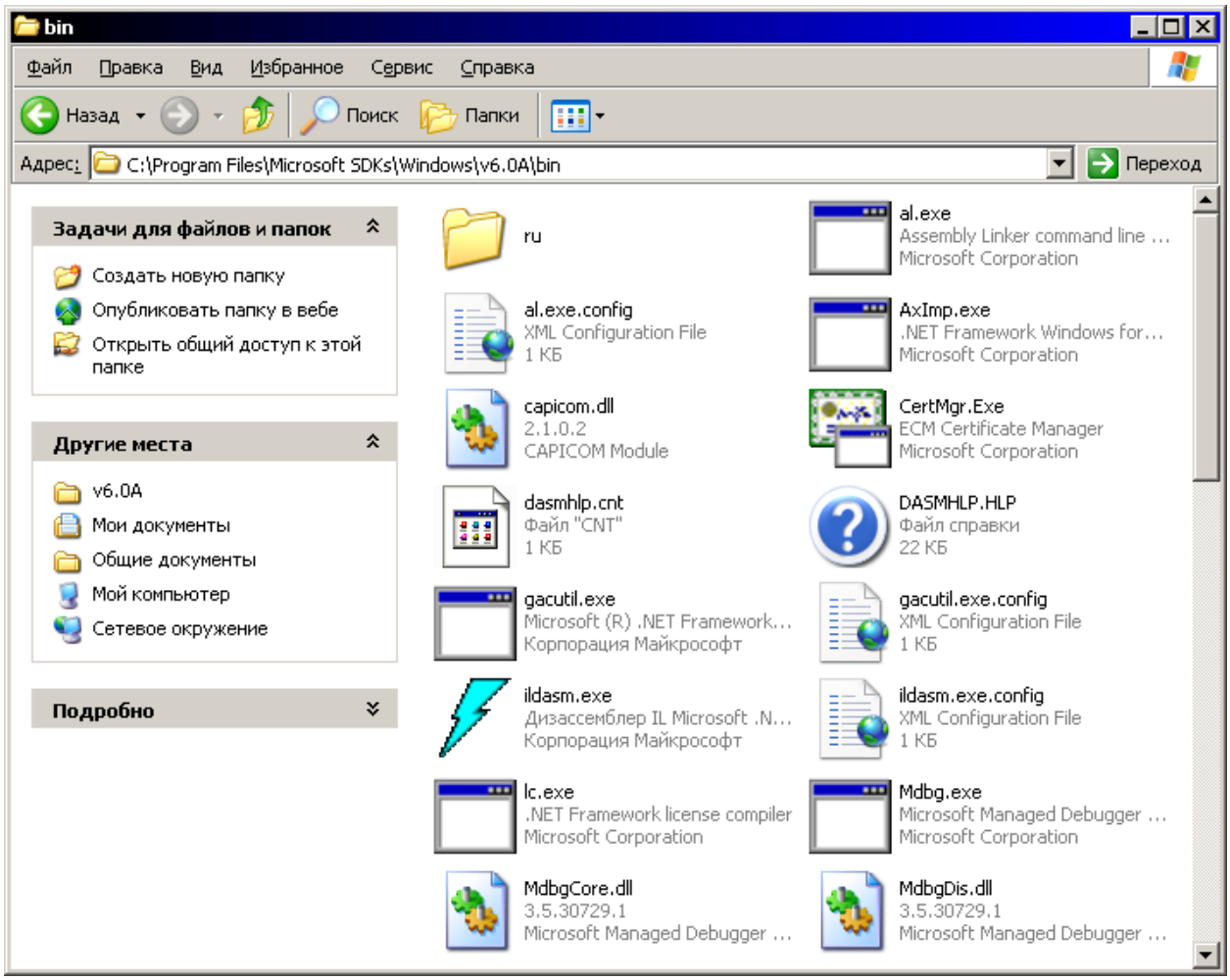


Рис. 2.19 – Расположение программы ILDASM

• Ввести в командной строке или в меню запуска программы (Win+R) «ildasm». Если пути к папке SDK прописаны в системе, то дизассемблер бу-

дет запущен. Если нет, следует воспользоваться первым вариантом.

Во-вторых, в среде Visual Studio существует возможность добавлять новые инструменты в меню «Сервис». Для этого следует выбрать пункт меню «Сервис → Внешние инструменты…». Появится диалог, изображенный на рисунке 2.20. Нажимаем кнопку «Добавить», придумываем название для нового пункта меню (амперсандом можно пометить символ-акселератор), указываем путь к файлу ildasm.exe и исходный каталог (просто копируем название каталога из поля «Команда»). Ставим галочку «Использовать окно вывода» (чтобы при запуске ILDASM не появлялось консольное окно). После этого нажимаем «OK», и в меню «Сервис» появляется новый пункт «MSIL Disassembler».

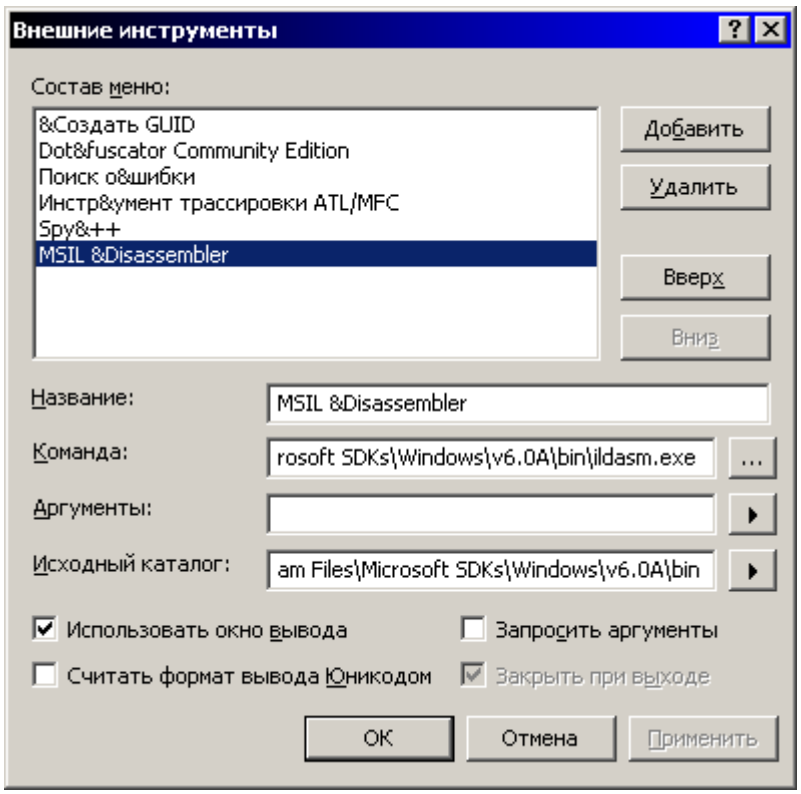


Рис. 2.20 – Добавление нового внешнего инструмента

Запустив ILDASM (рис. 2.21), выберем в меню «Файл» выберите команду «Открыть». В появившемся диалоговом окне откроем папку с приложением Sample_2_3_text.exe (хотя можно взять любую другую версию), которое мы создали чуть раньше, и выберем его.

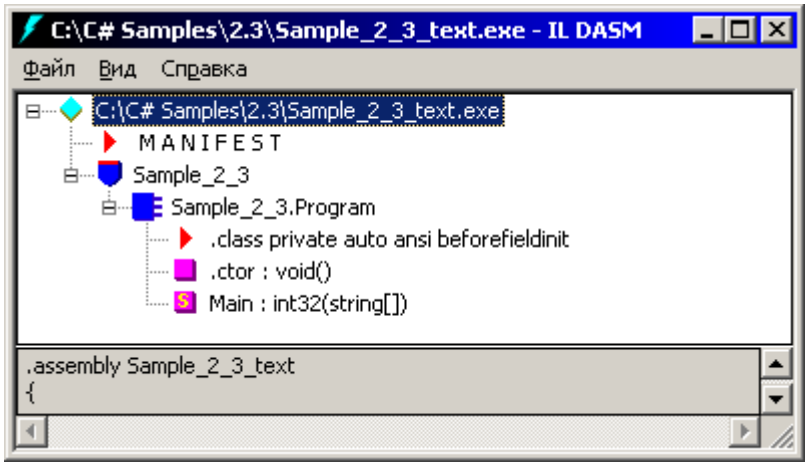


Рис. 2.21 – Главное окно ILDASM

В главном окне появится древовидное представление управляемого двоичного кода. В файле справки можно посмотреть значки, которыми ILDASM помечает те или иные компоненты .NET-приложения (рис. 2.22).

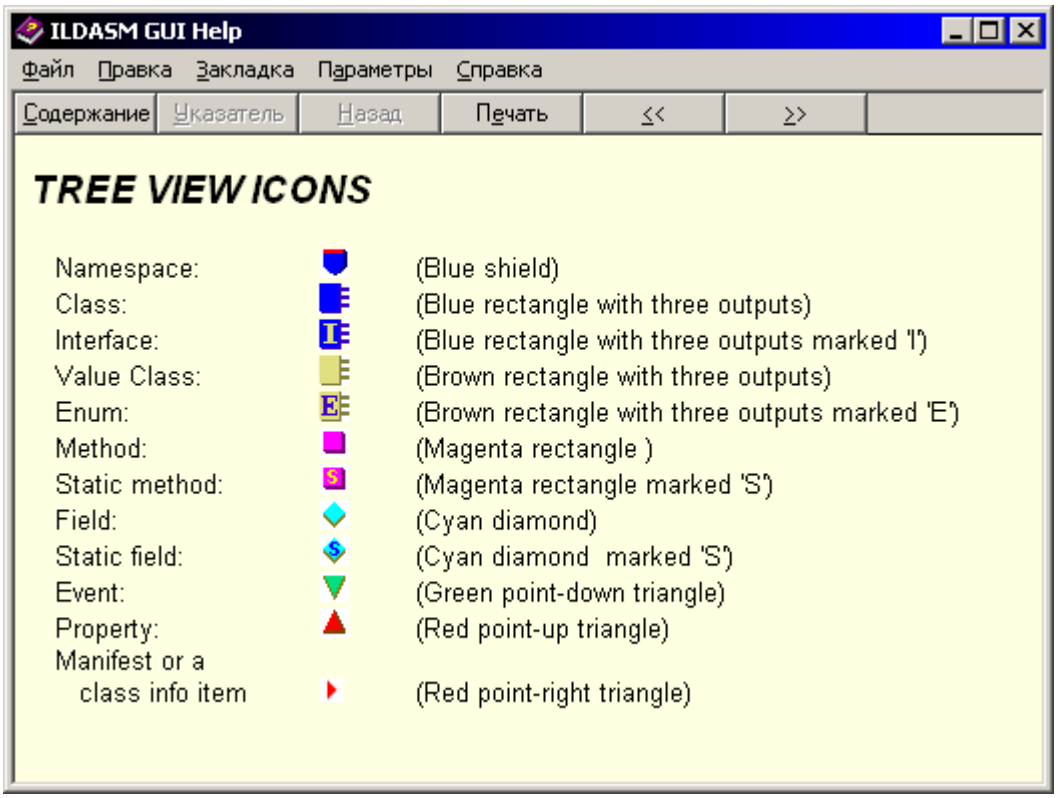


Рис. 2.22 – Условные обозначения ILDASM

Сравнив эти значки с представлением приложения «Hello, World!» в ILDASM, можно заметить, что приложение состоит из декларации, одного пространства имен (Sample_2_3), одного класса (Program), а также двух методов (конструктора класса и статического метода Main) и кое-какой инфор-

мации о классе.

В приложении «Hello, World!» наиболее интересен метод Main. Дважды щелкните значок метода Main в древовидной структуре приложения, и ILDASM выведет окно, отображающее MSIL-код метода Main (рис. 2.23).

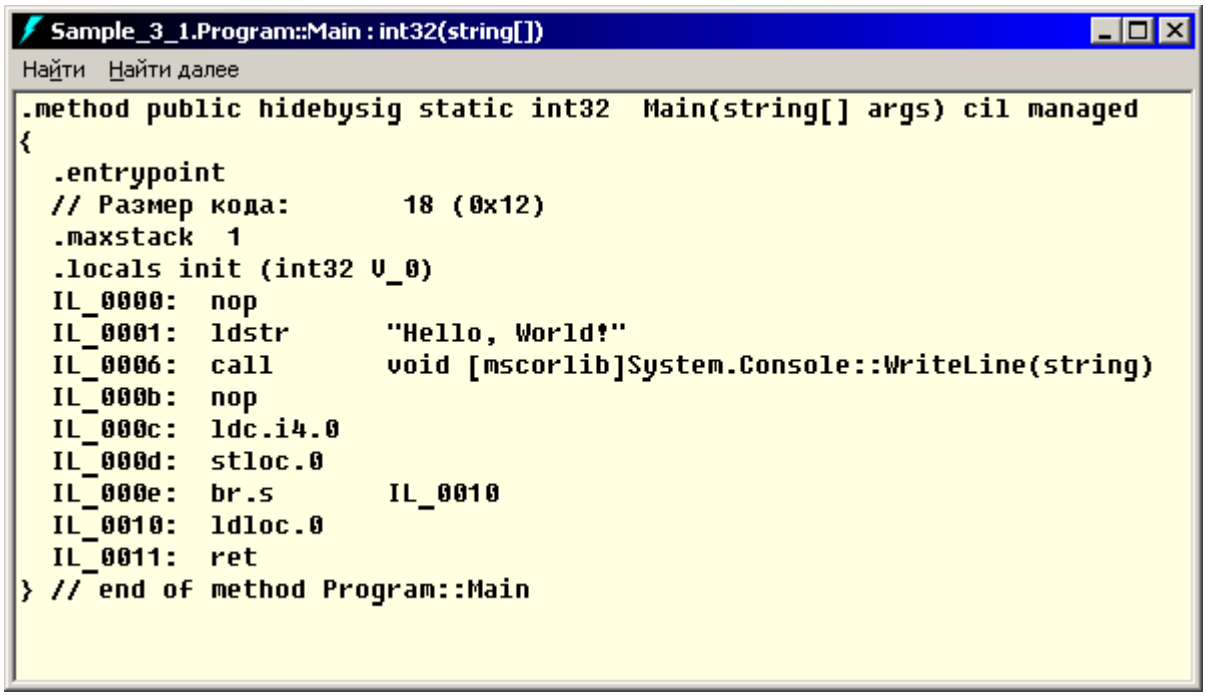


Рис. 2.23 – MSIL-код метода Main

В этом коде можно видеть кое-что, характерное для любого .NET-приложения:

```
.method public hidebysig static int32 Main(string[] args) cil managed
{
  .entrypoint
  // Размер кода:       18 (0x12)
  .maxstack 1
  .locals init (int32 V_0)
  IL_0000: nop
  IL_0001: ldstr        "Hello, World!"
  IL_0006: call         void [mscorlib]System.Console::WriteLine(string)
  IL_000b: nop
  IL_000c: ldc.i4.0
  IL_000d: stloc.0
  IL_000e: br.s         IL_0010
  IL_0010: ldloc.0
  IL_0011: ret
} // end of method Program::Main
```

Первая строка содержит описание метода Main с помощью ключевого слова .method. Можно заметить, что в этом описании есть модификаторы **public** и **static**, с которыми был описан метод Main. Кроме того, метод имеет атрибут managed (управляемый код). Это важная отличительная особенность,

так как на C# можно создать и «неуправляемый» (unmanaged), или «небезопасный» (unsafe), код.

Ключевое слово .entrypoint в следующей строке MSIL-кода указывает на то, что данный метод является точкой входа приложения. Когда исполняющая среда запускает приложение, управление программой передается коду, следующему за этой точкой.

Вызывают интерес и исполняемые коды в строках IL_0001 и IL_0006. В первой команда ldstr (load string) загружает в стек неизменяемый литерал («Hello, World!»). В следующей строке вызывается метод System.Console::WriteLine. Заметьте: к имени метода добавлено имя сборки (assembly), в которой описан этот метод (mscorlib). Такой уровень детализации MSIL хорош тем, что вы без труда напишете утилиту, которая выявит связи в программе и отобразит информацию о файлах, требующихся для правильной работы приложения. Кроме того, вы можете определить число аргументов метода и их типы. В нашем случае метод System.Console.WriteLine принимает объект System.String, который перед вызовом метода должен быть помещен в стек. И, наконец, в строке IL_0011 находится исполняемый MSIL-код ret – код возврата из метода.

Чтобы выяснить, какой код находится в файле EXE или DLL – управляемый или нет, попробуйте открыть его в ILDASM. Файл, содержащий MSIL-код и декларацию, будет открыт. В ином случае вы получите сообщение об ошибке (рис. 2.24).

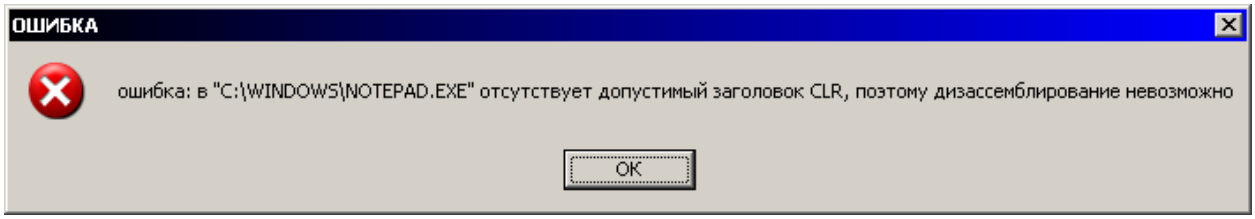


Рис. 2.24 – Ошибка при открытии файла без заголовка CLR

2.3.7. Файлы с примерами

Пример к § 2.3, набранный в текстовом редакторе (вместе с пакетным файлом, необходимыми для его компиляции) находится в файле Samples\2.3\Sample_2_3_text.cs. Пакетный файл называется cs23.bat. Если на вашей машине среда .Net Framework установлена в другой папке, нежели у автора, потребуется модификация этого файла. Файл cs.bat позволяет осуще-

ствить компиляцию программы на языке C# в консоли, указав имя файла с программой в качестве параметра.

Решение для SharpDevelop с примером к данному разделу располагается в папке Samples\2.3\Sample_2_3_spec. Структура решения во многом совпадает с таковой у Visual Studio 2008 – файл решения имеет имя Sample_2_3_spec.sln, а файл с программой – Program.cs. Решение для Visual Studio 2008 располагается в папке Samples\2.3\Sample_2_3_msvs.

В дальнейшем все примеры будут разрабатываться в Visual Studio 2008. Однако ничто не мешает компилировать их с консоли (правильно указав параметры компиляции) или в других совместимых средах. Даже если в используемой среде формат решения (и/или проекта) будет другим, можно создавать в их рамках пустые проекты и добавлять к ним файлы с программой Program.cs из решений для Visual Studio.

§ 2.4. Среда разработки

Рассмотрим основы работы в среде разработки Microsoft Visual Studio 2008.

2.4.1. Организация проекта

По умолчанию все файлы и папки пустого проекта, назовем его «Project», располагаются в папке «Project» (если не создавался отдельный каталог для решения). Это:

1. Project.sln – файл решения. Содержит список проектов решения, конфигурации построения решения и т.п.

2. Project.suo – пользовательские настройки решения. В данный файл записываются все параметры, связанные с решением, так что каждый раз при открывании решения включаются произведенные пользователем настройки (в т.ч. положение окон и диалогов, открытые файлы и т.д.).

3. Project.csproj – настройки проекта (задаваемые в диалоге свойств проекта, меню «Проект» → «Свойства: Project…», где Project – имя текущего проекта).

4. Program.cs – исходный файл с программой на языке C#.

5. Properties\AssemblyInfo.cs – файл на языке C#, содержащий сведения о сборке.

6. Папки bin, obj. Создаются при компиляции. Содержат исполняемые файлы, динамические библиотеки (в зависимости от типа проекта), различные вспомогательные файлы компиляции и компоновки. В принципе, их можно удалять, при повторном построении решения компилятор их сгенерирует снова.

Все исходные файлы и ссылки сборки можно видеть в окне обозревателя решений (рис. 2.25).

Здесь можно исключить файлы из проекта или удалить их, переименовать или открыть в редакторе кода. Двойной щелчок мышью по файлу с исходным кодом открывает его в редакторе. В файле AssemblyInfo.cs можно настроить некоторые свойства сборки – заголовок, описание, копирайт, версия и т.д. Все настройки имеют вид

```
[assembly: <класс>(<параметры>)]
```

Здесь класс – это класс соответствующего атрибута сборки. Об атрибутах мы будем говорить позже, в § 5.4.

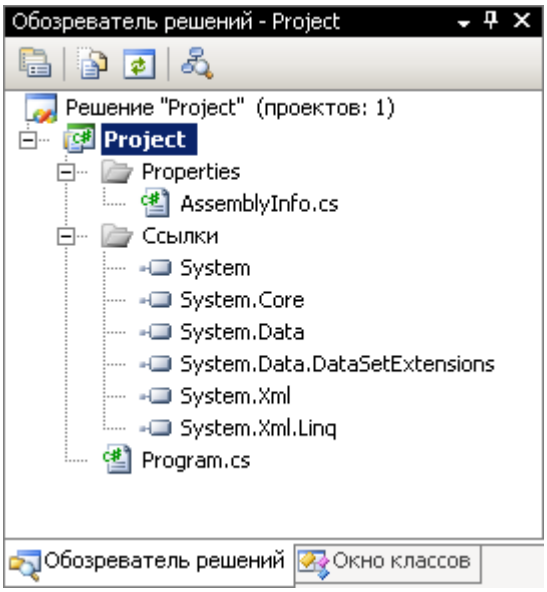


Рис. 2.25 – Обозреватель решений

Ссылки – это ссылки на сборки, доступные в проекте. Попытка использовать класс, структуру или другой тип, не включенный ни в одну из перечисленных сборок, приведет к ошибке компиляции. Подробнее об этом смотрите п. 4.1.3.

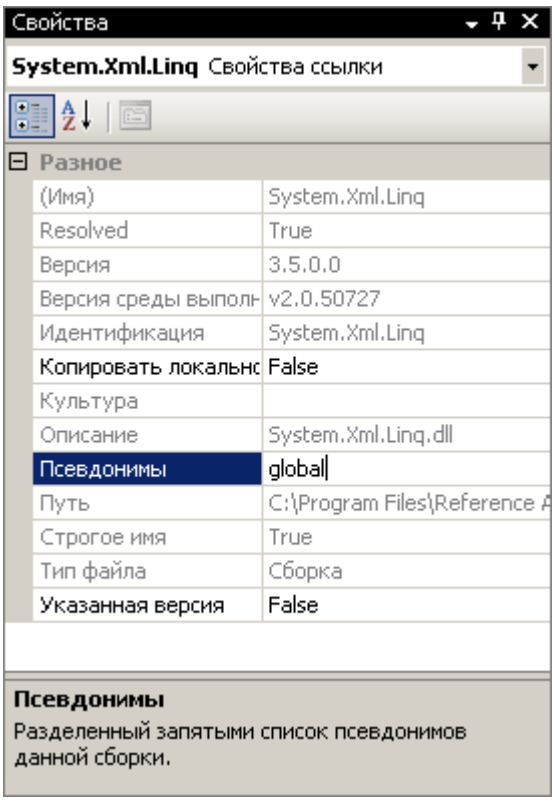


Рис. 2.26 – Свойства объектов

Ниже расположено окно свойств объектов (рис. 2.26). Здесь отображаются свойства любого выбранного объекта – решения, проекта, ссылки, исходного файла, компонента в редакторе форм и т.д. Выбрав какой-либо элемент в обозревателе решений, мы можем увидеть его свойства. Серый шрифт используется для свойств, доступных только для чтения, остальные свойства можно изменять. Так, можно изменять имена файлов, действия при построении, псевдонимы ссылок и т.д.

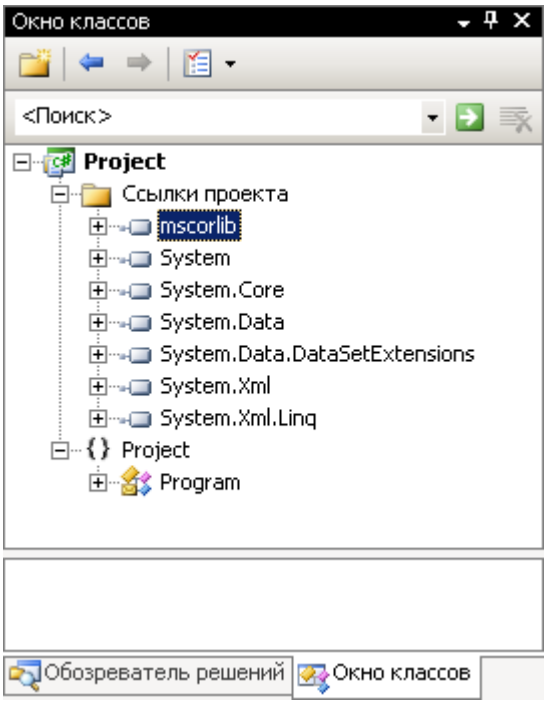


Рис. 2.27 – Окно классов

Переключившись на закладку «Окно классов», можно посмотреть, какие типы описаны в подключенных сборках (а также в сборке mscorlib, которая подключается по умолчанию), а также какие типы описаны в самом проекте (рис. 2.27).

2.4.2. Редактор кода

В редакторе кода для каждого файла создается отдельная закладка (рис. 2.28). Выпадающие списки в верхней части редактора можно использовать для быстрого перехода к описанию типа и его члена.

Код организовать в виде иерархического дерева, отдельные ветви которого можно сворачивать и разворачивать. Можно описывать свои сворачива-

емые области кода (см. § 3.6).

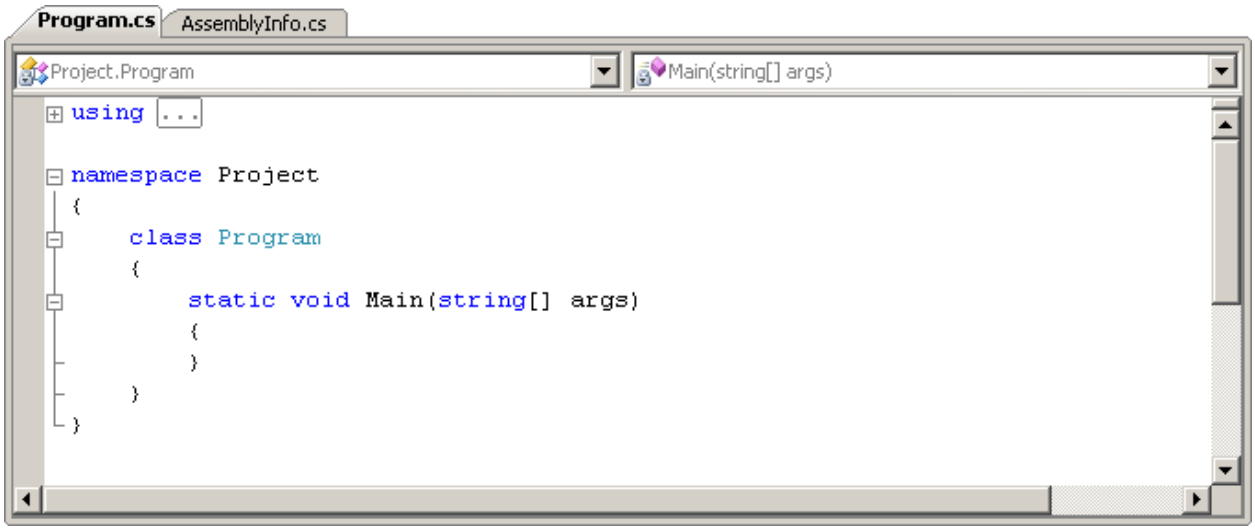


Рис. 2.28 – Редактор кода

Доступны стандартные средства редактирования текста – копирование (Ctrl+C или Ctrl+Insert), вставка (Ctrl+V или Shift+Insert), вырезание (Ctrl+X или Shift+Del), удаление (Del), поиск (Ctrl+F, F3 для повторного поиска), замена (Ctrl+H), отмена (Ctrl+Z или Alt+BS), возврат отмененного действия (Ctrl+Y или Alt+Shift+BS). Если выделение отсутствует, команда Ctrl+C копирует, а Ctrl+X вырезает в буфер всю текущую строку. Для удаления фрагмента от текущей позиции и до конца лексемы используется комбинация клавиш Ctrl+Del.

Набор кода происходит в режиме автозавершения ввода. Т.е. когда мы начинаем набор, среда разработки предлагает варианты завершения конструкции. Для вызова списка автоматического завершения также используется комбинация клавиш Ctrl+Пробел (если, например, мы этот список закрыли, но хотим отобразить вновь). Если среди предлагаемых вариантов нет нужного – значит, в программе допущена ошибка. Например, не подключено пространство имен, в котором описан требуемый тип. Или происходит попытка описать метод вне класса, вызвать статический член класса через его экземпляр и т.д.

При вызове метода появляется подсказка о типе и количестве формальных параметров вызова метода. Если метод имеет несколько полиморфных реализаций, можно их пролистывать, выбирая нужную (рис. 2.29). Если окно с подсказкой закрылось, можно отобразить его вновь, используя комбинацию клавиш Shift+Ctrl+Пробел.

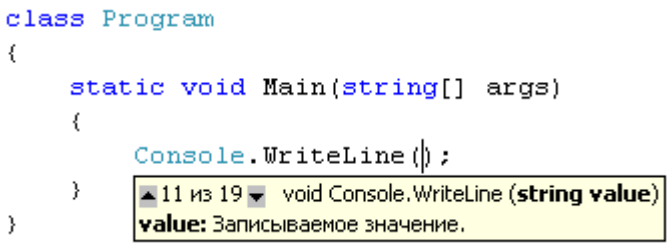


Рис. 2.29 – Параметры метода

Также компилятор автоматически форматирует выражения, расставляя пробелы между лексемами и отступы блоков.

Есть некоторые полезные инструменты в меню «Оптимизация кода» – переименование типа или метода, оформление части локального кода в виде метода, преобразование локальной переменной в параметр и т.д. Переименование предлагается редактором автоматически. Когда мы меняем имя класса, метода и других элементов программы, появляется специальный маркер, при наведении на который можно осуществить переименование или переименование с предварительным просмотром результата (рис. 2.30).

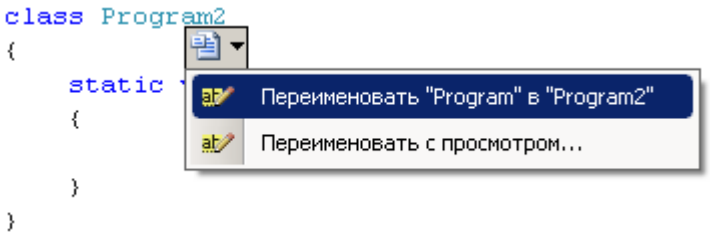


Рис. 2.30 – Переименование класса

При помощи кнопок на панели инструментов или подпунктов меню «Правка» → «Дополнительно» можно заключить фрагмент кода в комментарий или выполнить обратное преобразование, уменьшить или увеличить отступ блока кода.

В меню «Правка» → «IntelliSence» имеются более продвинутые инструменты управления кодом (создание заглушек методов, реализация интерфейсов, краткие сведения о типе и т.д.).

Краткие сведения о типе отображаются автоматически в виде всплывающей подсказки при наведении указателя мыши на наименование типа в коде. Но можно вызвать подсказку вручную, используя сочетание клавиш Ctrl+K+I (сначала нажать Ctrl+K, а затем, не отпуская Ctrl, нажать I).

Для вставки фрагмента наиболее часто используемого кода используется комбинация клавиш Ctrl+K+X. Например, для вставки в код вызова ме-

тода Console.WriteLine() нажимаем Ctrl+K+X, выбираем в появившемся списке разделов «Visual C#», и далее элемент «cw». Посмотреть имеющиеся фрагменты кода и добавить новые можно в диалоге «Диспетчер фрагментов кода» (меню «Сервис» → «Диспетчер фрагментов кода…» или комбинация клавиш Ctrl+K+B).

При вставке фрагмент кода замещает собой фрагмент, выделенный в настоящий момент. Если, например, нами уже набран код, и мы хотим заключить его в цикл, то используем размещение во фрагменте. Для этого выделяем требуемый код, нажимаем комбинацию клавиш Ctrl+K+S, и выбираем требуемый шаблон (for, foreach, forr и т.д.).

2.4.3. Встроенный отладчик

Если приложение содержит ошибки, то оно не будет построено и запущено. Среда разработки предложит запустить последний успешно построенный вариант, если такой есть (рис. 2.31).

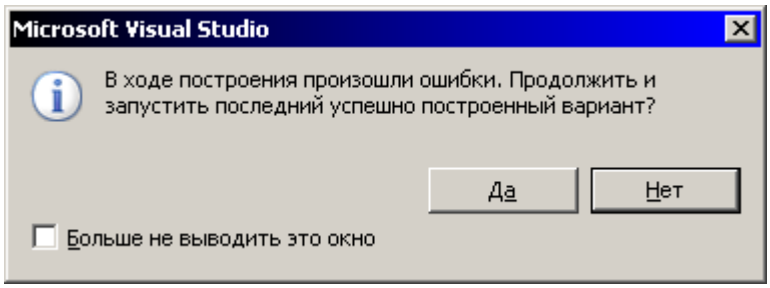


Рис. 2.31 – Ошибки при построении

В принципе, не обязательно запускать компиляцию для обнаружения ошибок. Среда разработки практически сразу выделяет их в тексте программы. Для исправления ошибки, если непонятно, почему она появляется, можно изучить описание ошибки в справочной системе, посмотреть правила синтаксиса используемой конструкции языка C# или изучить спецификации используемого типа.

На предупреждения компилятора также необходимо реагировать. Желательно, чтобы в программе их не было. Поэтому либо измените код программы, чтобы он не вызывал неоднозначных трактовок, либо используйте соответствующие директивы препроцессора (см. § 3.6).

Ошибки и предупреждения, детектируемые компилятором, имеют характер *ошибок и предупреждений времени разработки* (design-time). После того, как они устранены, программу можно запускать на выполнение. В про-

цессе работы программы возникают ошибки другого рода – *ошибки времени исполнения* (runtime errors). Для их выявления и исправления используется *отладка*. Отладка заключается в отслеживании текущих значений локальных переменных и полей типов, а также выяснении путей передачи управления в программе. Для выяснения путей передачи управления производятся такие действия, как *просмотр стека вызовов* и *трассировка*.

Нам известно, что при вызове метода B из метода A текущий контекст метода A сохраняется на стеке, чтобы после завершения работы метода B можно было вернуться к выполнению следующей команды метода B. Мы здесь не говорим о процедурах и функциях, т.к. язык C# является чисто объектным языком. Также под методами подразумеваются конструкторы, методы доступа к свойствам и т.п. В контекст метода входят адрес возврата, значения локальных переменных и параметров метода. Изучая стек вызовов, можно понять, в каком порядке вызывались методы и с какими параметрами. Особенно это полезно при отладке оконных приложений, когда некоторые методы (события) вызываются диспетчерами сообщений, а не из кода программы. Также стек вызовов дает ценную информацию при отладке рекурсивных вызовов.

Трассировка заключается в возможности выполнения программного кода не целиком, а по шагам. Обычно трассировка начинается с какой-либо *точки останова*. Точка останова – это помеченная специальным образом строка кода, дойдя до выполнения которой среда исполнения прервет выполнение программы. В этот момент изучаются состояние стека вызовов, значения переменных и полей на текущий момент. После этого можно продолжить выполнение программы либо до следующей точки останова, либо только до следующей строки.

При отладке можно использовать инструменты CLR или отладчик Microsoft Visual Studio.

2.4.3.1. Отладчик CLR

Для того, что бы можно было отлаживать приложение средствами SDK при сборке проекта из командной строки нужно использовать ключи о включении отладочной информации (табл. 2.1).

Табл. 2.1 – Ключи компилятора для управления отладочной информацией

Ключ	Описание
`/debug-`	Нет отладочной информации
`/debug` `/debug+`	Отладочная информация будет добавлена
`/debug:pdbonly`	Создавать PDB файл с отладочной информацией
`/debug:full`	Создать PDB файл и дополнительные отладочные сведения (DebuggableAttribute) для компилятора JIT

По умолчанию отладочная информация не включается, а если мы ее добавляем, но не указываем тип, используется «full».

Итак, в папке «Samples\2.4» создадим файл «Sample_2_4.cs», поместим в него текст программы «Hello, World!» и откомпилируем его с отладочной информацией. Командную строку компиляции поместим в файл «cs24.bat»:

```
csc.exe /debug+ Sample_2_4.cs
```

После компиляции появился исполняемый файл «Sample_2_4.exe» и файл с отладочной информацией «Sample_2_4.pdb».

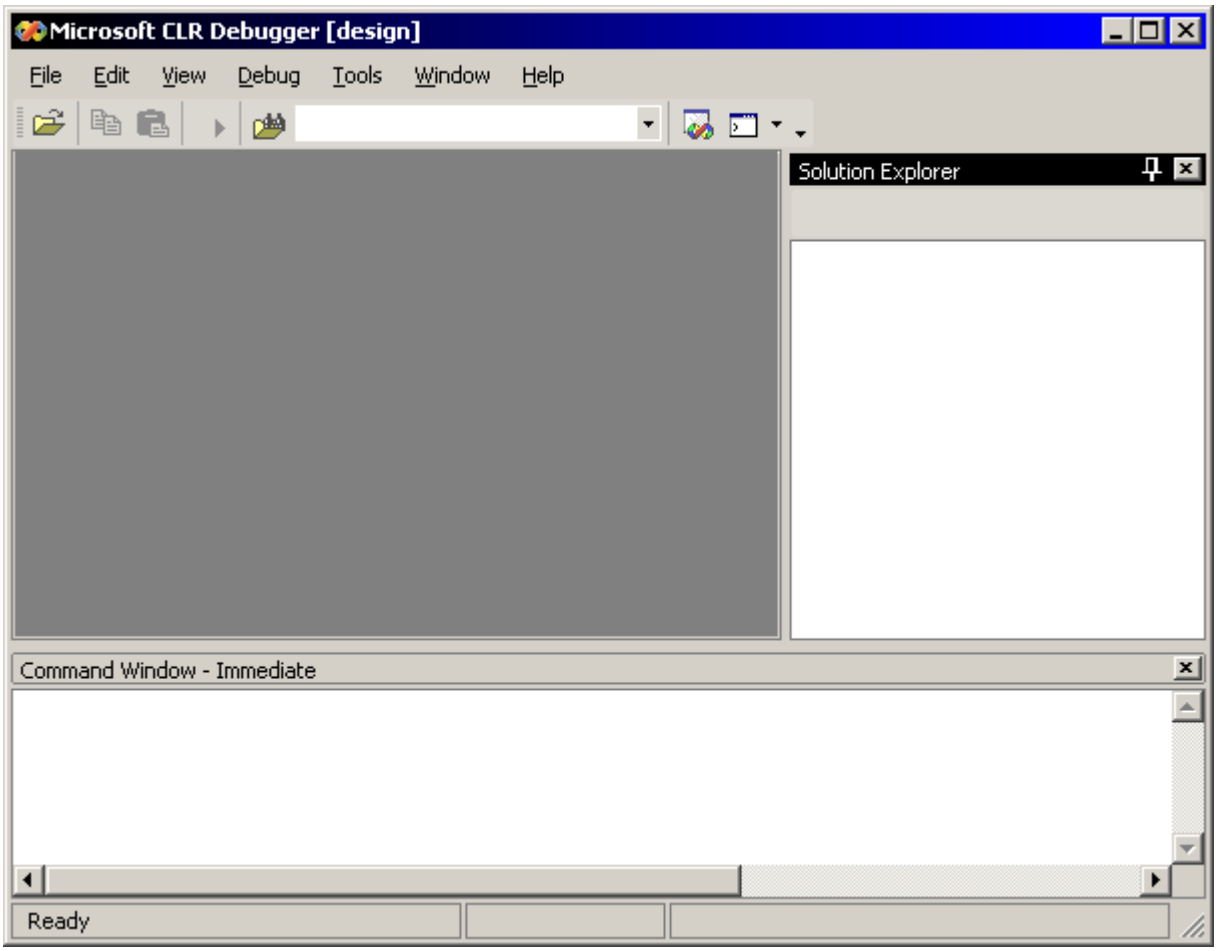


Рис. 2.32 – Инструмент CLR Debugger

После этого можно использовать отладчик CLR, размещающийся по следующему пути:

```
<папка Program Files>\Microsoft.NET\SDK\<версия>\GuiDebug\DbgCLR.exe
```

Для этого на компьютере должен быть установлен пакет .NET Framework SDK версии не выше 2.0 (т.е. 1.0, 1.1 или 2.0). В отличие от среды выполнения .NET Framework, пакет SDK (Software Development Kit) содержит подробную документацию и примеры программ, а также средства для тестирования и развертывания приложений. Пакет .NET Framework SDK версии 1.1 имеет размер около 100 Мб, версии 2.0 – около 350 Мб. Скачать их можно из центра загрузок Microsoft (microsoft.com/downloads/ru-ru).

На рис. 2.32 изображен интерфейс программы CLR Debugger для .NET Framework SDK 1.1.

Для начала отладки приложения выбираем пункт меню «Debug» → «Program To Debug…». Появляется диалог выбора программы (рис. 2.33). Здесь можно выбрать саму программу, аргументы командной строки и рабочий каталог. Рабочий каталог будет являться текущим после запуска приложения. Это важно, например, при работе с файлами. Если мы при чтении или записи файла указываем его относительную спецификацию, например, «files\1.txt» или «..\dir2\file.bin», то указанные спецификации будут отталкиваться от текущего каталога приложения. Т.е. полные спецификации будут выглядеть так:

```
<рабочий каталог>\files\1.txt
<рабочий каталог>\..\dir2\file.bin
```

По умолчанию рабочим является каталог, в котором расположен исполняемый файл.

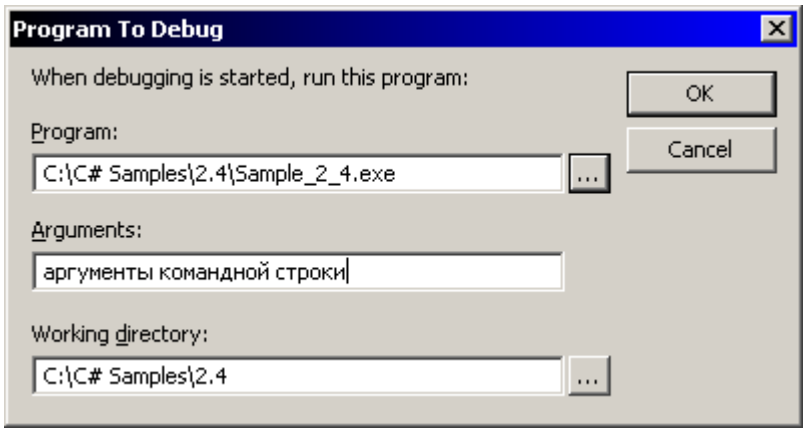


Рис. 2.33 – Выбор программы

Затем открываем исходные файлы (меню «File»→ «Open…», соответствующая кнопка на панели инструментов или комбинация клавиш Ctrl+O) (рис. 2.34).

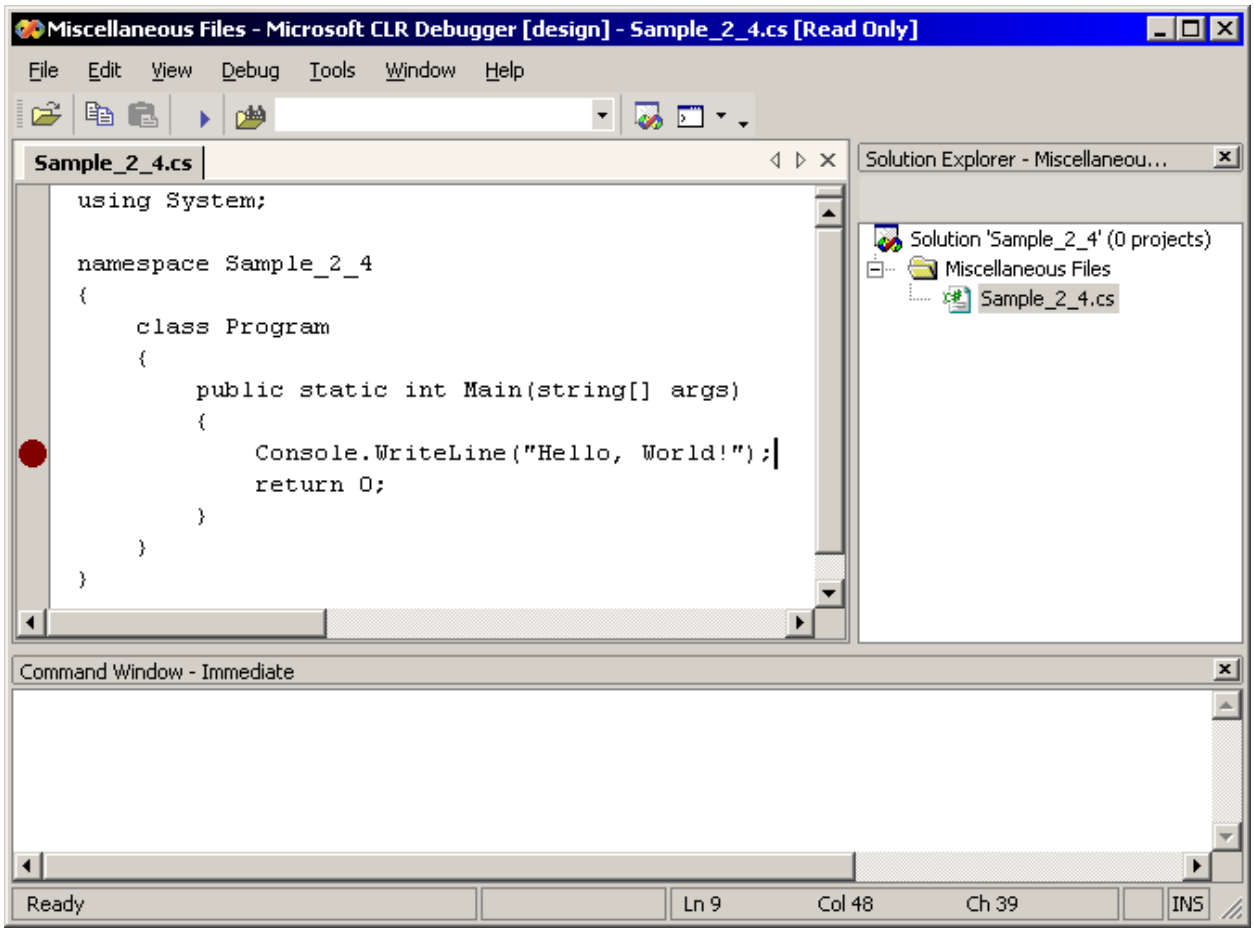


Рис. 2.34 – Отладка в CLR Debugger

Далее доступны все те же действия, что и при отладке из Visual Studio. Поэтому подробно рассматривать CLR Debugger мы не будем. Этим инструментом ранее пользовались разработчики, не имевшие лицензионной версии Visual Studio. Однако, в настоящее время доступны бесплатные версии Visual Studio Express.

2.4.3.2. Отладчик Visual Studio

Возможности отладчика Visual Studio соответствуют возможностям отладчика CLR. При трассировке наиболее часто используются следующие команды:

• Переход на следующий шаг с заходом в вызываемые методы («Отладка» → «Шаг с заходом», F11);

• Переход на следующий шаг без захода в вызываемые методы («Отладка» → «Шаг с обходом», F10);

• Продолжить выполнение программы до конца текущего метода («Отладка» → «Шаг с выходом», Shift+F11);

• Продолжить выполнение программы до конца или следующей точки останова («Отладка» → «Продолжить», F5);

• Прервать выполнение программы («Отладка» → «Остановить отладку», Shift+F5);

• Остановить выполнение программы и перевести ее в состояние трассировки («Отладка» → «Приостановить все», Ctrl+Alt+Break). Это полезно, например, если программа зациклилась, но где – неизвестно;

• Установить или убрать точку останова («Отладка» → «Точка останова», F9). Для создания или удаления точки останова можно также щелкнуть мышью по серому полю в левой части окна кода напротив интересующей строки кода.

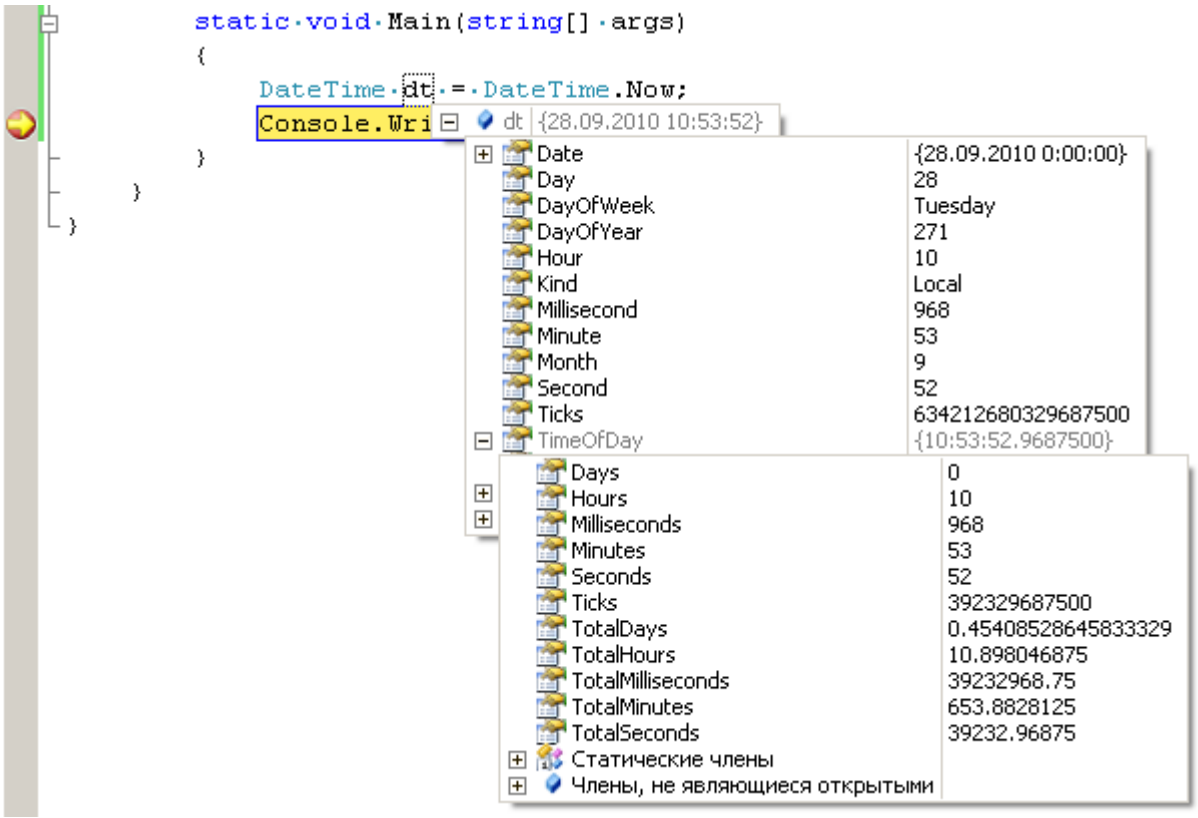


Рис. 2.35 – Отладка в Visual Studio

Точка останова помечается на сером поле красным кругом, текущая строка при отладке – желтой стрелкой, а строка, на которой было остановле-

но выполнение программы – зеленой стрелкой. Строка кода, для которой установлена точка останова, отображается на красном фоне; текущая конструкция при отладке – на желтом фоне; конструкция, на которой было остановлено выполнение программы – на зеленом фоне (рис. 2.35).

Все эти действия доступны только из среды разработки. Дополнительные команды смотрите в меню «Отладка». Прервать выполнение программы из консоли позволяет нажатие комбинации клавиш Ctrl+C или закрытие окна консоли.

Для наблюдения за значениями локальных переменных, а также полей и свойств типов достаточно навести курсор мыши на экземпляр типа. Если тип содержит вложенные экземпляры, то их также можно разворачивать для просмотра (рис. 2.35).

Также при отладке отображаются окна с дополнительной информацией, в том числе:

• «Видимые» – обзор значений видимых переменных, полей и свойств;

• «Локальные» – обзор значений локальных переменных;

• «Точки останова» – информация об имеющихся точках останова. Здесь можно некоторые точки удалить или отключить временно, а также задать условия останова;

• «Стек вызовов» – содержимое стека вызовов.

Например, на рис. 2.36 среда разработки настроена так, что отображаются окна «Видимые» и «Стек вызовов».

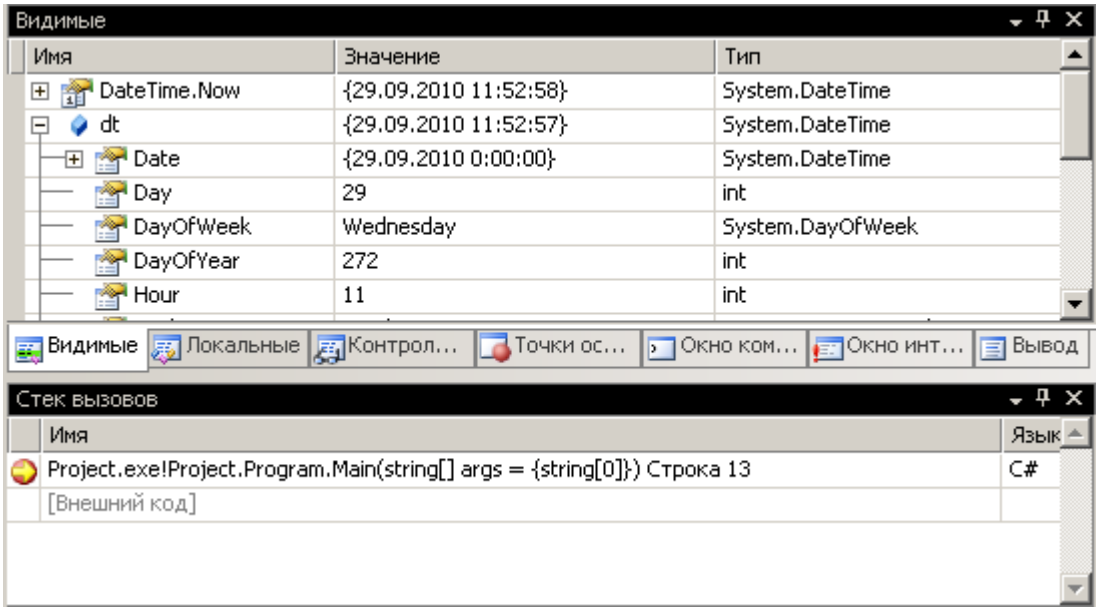


Рис. 2.36 – Обзор значений переменных и стека вызовов

При возникновении в программе необработанной исключительной ситуации, среда разработки сигнализирует об этом, показывая окно с информацией об исключении и месте его появления в программе (рис. 2.37).

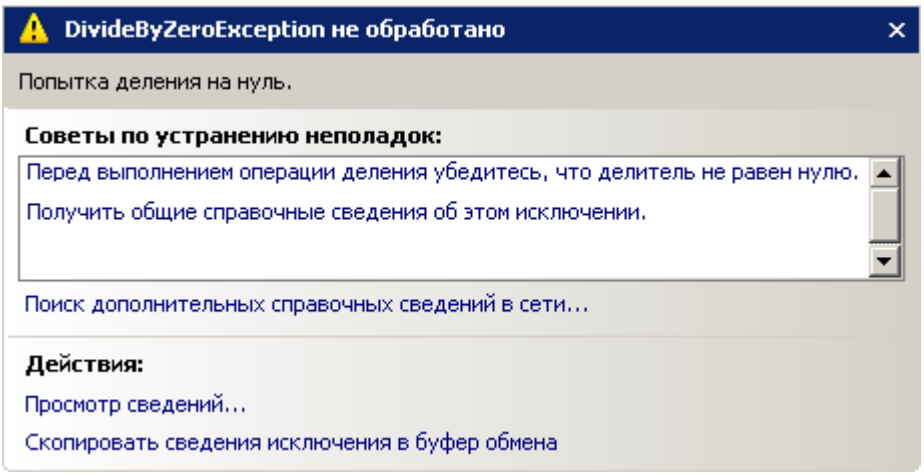


Рис. 2.37 – Необработанная исключительная ситуация

Если же программа была запущена без отладки (Ctrl+F5) или не из среды разработки, то отобразится окно с информацией об ошибке (рис. 2.38). При нажатии на кнопку «ОК» информация об исключении будет выведена на консоль, и затем приложение будет закрыто. При нажатии на кнопку «Отмена» будет запущен отладчик Visual Studio.

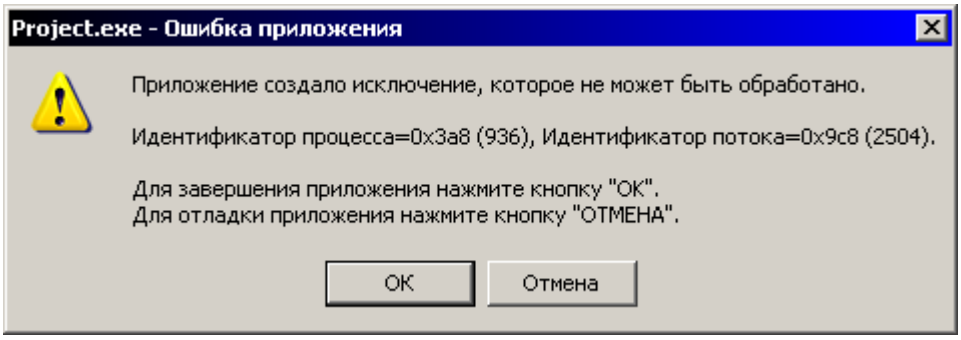


Рис. 2.38 – Ошибка приложения

После исправления всех ошибок времени исполнения в приложении можно построить его финальную версию (Release), выбрав соответствующий пункт в выпадающем списке на панели инструментов или в свойствах проекта («Проект» → «Свойства…», закладка «Построение»).

3. ОСНОВЫ ЯЗЫКА C#

У нас уже есть простая, но работающая программа. Можно приступать к изучению конструкций языка C#, добавляя их к этой программе.

Рассмотрим также алфавитный список ключевых слов языка C# с кратким указанием их функций и разделов учебного пособия, в которых о них говорится более подробно (табл. 3.1).

Табл. 3.1 – Список ключевых слов языка C#

КС	Описание	Пункт
abstract	Абстрактный класс или член класса	4.2.1-2
as	Преобразование типов данных	3.3.3.7
base	Доступ к конструктору базового класса Доступ к члену базового класса	4.4.2 4.7.2.2
bool	Логический тип данных	3.1.2.3
break	Завершение цикла Завершение оператора выбора	3.4.4.1 3.4.2.2
byte	Целый тип данных	3.1.2.1
case	Ветвление оператора выбора	3.4.2.2
catch	Обработка исключительной ситуации	3.4.5.2
char	Символьный тип данных	3.1.2.4
checked	Включение проверки переполнения для целых типов	3.3.3.10
class	Описание класса Ограничение параметров типа	4.2 5.1.2
const	Описание неизменной локальной переменной Описание неизменного члена класса	3.1.6.3 4.3.1
continue	Переход на следующую итерацию цикла	3.4.4.2
decimal	Тип данных с плавающей точкой	3.1.2.2
default	Метка по умолчанию в операторе выбора Значение типа по умолчанию	3.4.2.2 3.1
delegate	Описание делегата	4.8.2
do	Оператор цикла	3.4.3.2
double	Тип данных с плавающей точкой	3.1.2.2
else	Часть условного оператора	3.4.2.1
enum	Объявление перечисления	3.1.2.5

event	Объявление события Описание атрибута события	4.8.3 5.4.1
explicit	Объявление оператора явного преобразования типов	4.7.3.4
extern	Объявление метода с внешней реализацией Описание внешнего псевдонима	4.2.2 4.1.3.2
false	Логическая константа «ложь» Логический оператор	3.1.2.3 3.3.3.3
finally	Завершение блока обработки исключительной ситуации	3.4.5.2
fixed	Объявление неперемещаемой переменной или буфера	5.5.2.3
float	Тип данных с плавающей точкой	3.1.2.2
for	Оператор цикла	3.4.3.3
foreach	Оператор цикла	3.4.3.4
goto	Переход на ветку оператора выбора Переход на именованную метку	3.4.2.2 3.4.4.3
if	Условный оператор	3.4.2.1
implicit	Объявление оператора неявного преобразования типов	4.7.3.4
in	Часть оператора цикла foreach Часть выражения запроса LINQ	3.4.3.4 –
int	Целый тип данных	3.1.2.1
interface	Объявление интерфейса	4.9.1
internal	Модификатор доступа	4.2.1-2
is	Проверяет совместимость объекта с заданным типом	3.3.3.9
lock	Блокировка фрагмента кода	5.2.3.2
long	Целый тип данных	3.1.2.1
namespace	Объявление пространства имен	4.1.1
new	Оператор создания экземпляра объекта Сокрытие члена базового класса Ограничение типов в универсальном объявлении Часть выражения запроса LINQ	3.3.3.1 4.2.1-2 5.1.2 –
null	Пустая ссылка	3.1.3
object	Псевдоним класса Object	3.1.3
operator	Перегрузка встроенного оператора	4.7.3.1
out	Передача аргумента в метод по ссылке	4.4.1.1
override	Перегрузка абстрактного/виртуального члена класса	4.6.2.2

params	Произвольное количество аргументов метода	4.4.1.3
private	Модификатор доступа	4.2.1-2
protected	Модификатор доступа	4.2.1-2
public	Модификатор доступа	4.2.1-2
readonly	Модификатор доступа	4.3.2
ref	Передача аргумента в метод по ссылке	4.4.1.1
return	Прерывание выполнения метода Описание атрибута возвращаемого значения	3.4.4.4 5.4.1
sbyte	Целый тип данных	3.1.2.1
sealed	Описание изолированного класса или члена класса	4.2.1-2
short	Целый тип данных	3.1.2.1
sizeof	Размер типа данных по значению	3.3.3.9
stackalloc	Выделение блока памяти в стеке	5.5.2.4
static	Объявление статического класса или члена класса	4.2.1-2
string	Псевдоним класса String	3.1.3.1
struct	Объявление структуры Ограничение параметров типа	3.1.2.6 5.1.2
switch	Оператор выбора	3.4.2.2
this	Ссылка на текущий экземпляр класса Объявление индексатора Описание метода расширения	4.2.3 4.5.2 4.4.1.2
throw	Генерация исключительной ситуации	3.4.5.2
true	Логическая константа «истина» Логический оператор	3.1.2.3 3.3.3.3
try	Блок обработки исключительных ситуаций	3.4.5.2
typeof	Получение метакласса типа	3.3.3.9
uint	Целый тип данных	3.1.2.1
ulong	Целый тип данных	3.1.2.1
unchecked	Выключение проверки переполнения	3.3.3.10
unsafe	Блок небезопасного (неуправляемого) кода	5.5.2.1
ushort	Целый тип данных	3.1.2.1
using	Использование пространства имен или псевдонима Оператор управления ресурсами	4.1.2 3.5.3
virtual	Объявление виртуального члена класса	4.7.2.1

void	Объявление метода, не возвращающего значение	4.4.1
volatile	Объявление поля, изменяемого несколькими потоками	5.2.3.5
while	Оператор цикла	3.4.3.1

Также есть ряд контекстно-зависимых ключевых слов (табл. 3.2). Т.е. они считаются таковыми только в определенных конструкциях языка.

Табл. 3.2 – Контекстно-зависимые ключевые слова языка C#

КС	Описание	Пункт
add	Метод добавления обработчика события	4.8.3
alias	Описание внешнего псевдонима	4.1.3.2
ascending	Часть выражения запроса LINQ	–
assembly	Описание атрибута сборки	5.4.1
descending	Часть выражения запроса LINQ	–
by	Часть выражения запроса LINQ	–
equals	Часть выражения запроса LINQ	–
field	Описание атрибута поля	5.4.1
from	Часть выражения запроса LINQ	–
get	Описание метода чтения значения свойства	4.5.1.1
global	Псевдоним глобального пространства имен	4.1.3.2
group	Часть выражения запроса LINQ	–
into	Часть выражения запроса LINQ	–
join	Часть выражения запроса LINQ	–
let	Часть выражения запроса LINQ	–
method	Описание атрибута метода	5.4.1
module	Описание атрибута модуля	5.4.1
on	Часть выражения запроса LINQ	–
orderby	Часть выражения запроса LINQ	–
param	Описание атрибута параметра	5.4.1
partial	Частичное определение объекта или метода	4.2.1-2
property	Описание атрибута свойства	5.4.1
remove	Метод удаления обработчика события	4.8.3
select	Часть выражения запроса LINQ	–
set	Описание метода записи значения свойства	4.5.1.1
type	Описание атрибута типа	5.4.1
value	Доступ к записываемому значению свойства	4.5.1.1

	Доступ к обработчику события	4.8.3
var	Объявление локальной переменной неявного типа	3.1.4
where	Ограничение типов в универсальном объявлении Часть выражения запроса LINQ	5.1.2 –
yield	Управление настраиваемым итератором	3.4.4.6

Вне указанных конструкций они ключевыми словами не считаются. Например, можно объявить переменную с именем «value», но только вне описания свойства или события.

§ 3.1. Типы данных. Идентификаторы

Все типы данных делятся на два больших класса – типы данных по значению и типы данных по ссылке. Существуют и указатели, но их можно использовать только в небезопасном коде, о них мы поговорим позже, в § 5.5. Отметим одно важное отличие языка C# от языка C++. В языке C++ конструкция, подобная следующей:

```
<тип> <имя переменной>;
```

объявляет новую переменную типа <тип> с указанным именем. Причем для переменной сразу же выделяется место в памяти на стеке, а если тип – это класс, то после этого вызывается конструктор данного класса. В языке C# этого не происходит! Память нужно выделять самостоятельно, используя оператор **new**:

```
<тип> <имя переменной> = new <тип>([<аргументы конструктора>]);
```

Существуют и другие способы инициализации переменных, о них мы поговорим ниже.

Каждый тип данных имеет значение по умолчанию (т.е. значение, принимаемое при инициализации конструктором по умолчанию, если он доступен):

• 0 для целых типов;

• 0.0 для чисел с плавающей точкой;

• **false** для булева типа;

• '\u0000' для символьного типа;

• 0 для перечисляемого типа (даже если константа с таким значением не определена);

• **null** для ссылочных типов;

• пустое значение для обнуляемых типов.

Для получения значения по умолчанию в явном виде также используется ключевое слово **default**:

```
<тип> <имя переменной> = default(<тип>);
```

3.1.1. Базовый класс System.Object

Все объекты .NET имеют единый базовый класс – System.Object. Соответственно, все типы данных (а они, очевидно, также являются классами)

наследуются от Object. И все методы, объявленные в классе Object, доступны для любого типа данных C#. Самые полезные из них – это:

```
public virtual string ToString();
public virtual bool Equals(object obj);
public static bool Equals(object objA, object objB);
public static bool ReferenceEquals(object objA, object objB);
public Type GetType();
```

Первый метод возвращает строковое представление класса. Этот метод является виртуальным, поэтому то, как он будет работать, зависит от логики классов-потомков.

Следующие два метода проверяют эквивалентность двух экземпляров классов .NET. Причем второй вариант метода, статический, реализован через первый, поэтому запись Object.Equals(A, B) эквивалентна записи A.Equals(B) (но не B.Equals(A)!). При перегрузке метода Equals также следует перегрузить метод GetHashCode (см. п. 4.7.3.5).

Пример, демонстрирующий перегрузку методов ToString и Equals:

```
class Five
{
	public override string ToString()
	{
		return "Five";
	}

	public override bool Equals(object obj)
	{
		return obj.ToString() == "Seven";
	}

	public override int GetHashCode()
	{
		return base.GetHashCode();
	}
}

class Seven
{
	public override string ToString()
	{
		return "Seven";
	}

	public override bool Equals(object obj)
	{
		return obj.ToString() == "Seven";
	}

	public override int GetHashCode()
	{
		return base.GetHashCode();
	}
```

```
    }

    static int Main(string[] args)
    {
        Five c5 = new Five();
        Seven c7 = new Seven();
        Console.WriteLine(c5);
        Console.WriteLine(c7);
        Console.WriteLine(c5.Equals(c7));
        Console.WriteLine(c7.Equals(c5));
        Console.WriteLine(Object.Equals(c5, c7));
        Console.WriteLine(Object.Equals(c7, c5));
        return 0;
    }
```

В данном примере мы пользуемся тем, что первый вариант метода Equals виртуальный, и перегружаем его (см. § 4.7). Также перегружаем методы ToString. Теперь класс Five считает, что он эквивалентен тому классу, чье строковое представление – «Seven». Класс Seven считает, что он также эквивалентен такому классу. В результате работы на консоль будет выведен следующий текст:

```
Five
Seven
True
False
True
False
```

Следовательно, в некоторых ситуациях запись A.Equals(B) не эквивалентна записи B.Equals(A).

Четвертый метод проверяет совпадение ссылок на объекты. Рассмотрим следующий пример:

```
int x1 = new int();
int y1 = x1;
Console.WriteLine("x1 = " + x1 + "; y1 = " + y1);
Console.WriteLine(x1.Equals(y1) ? "x1 == y1" : "x1 != y1");
Console.WriteLine(Object.ReferenceEquals(x1, y1) ?
    "ref x1 == ref y1" : "ref x1 != ref y1");

Object x2 = new Object();
Object y2 = x2;
Console.WriteLine("x2 = " + x2 + "; y2 = " + y2);
Console.WriteLine(x2.Equals(y2) ? "x2 == y2" : "x2 != y2");
Console.WriteLine(Object.ReferenceEquals(x2, y2) ?
    "ref x2 == ref y2" : "ref x2 != ref y2");
```

При копировании типов по значению копируются значения объектов, а при копировании ссылочных типов – только ссылки (об этом мы будем говорить далее в этом параграфе). Поэтому на консоли увидим:

```
x1 = 0; y1 = 0
```

```
x1 == y1
ref x1 != ref y1
x2 = System.Object; y2 = System.Object
x2 == y2
ref x2 == ref y2
```

Итак, по умолчанию целые переменные равны нулю (и имеют соответствующее строковое представление), а строковое представление экземпляров класса System.Object содержит имя данного класса. Поэтому x1 = y1 и x2 = y2. Но экземпляры класса **int** хранят свои данные в разных ячейках памяти на стеке, поэтому и ссылки на них не совпадают. А экземпляры класса Object ссылаются на один и тот же объект.

И последний из рассматриваемых методов, **пятый метод** класса Object, возвращает метакласс типа. Метаданные – это данные о данных, а метакласс, соответственно – это класс, инкапсулирующий сведения о другом классе. О метаданных мы будем говорить в § 5.3.

Пока мы еще не изучили язык C# в степени, достаточной для написания таких программ. Поэтому можно просто взять готовый для изучения проект.

 Пример: Samples\3.1\3_1_1_object.

3.1.2. Типы данных по значению

Отметим следующие особенности типов данных по значению:

1) Данные, как и в языке C++, хранятся на стеке;

2) Размещаются в памяти гораздо быстрее;

3) Автоматически удаляются при выходе из зоны видимости;

4) Только для данных фиксированной длины.

Типы по значению разделяются на простые типы, перечисляемые (Enum), обнуляемые (Nullable) типы и типы структуры. К простым типам относятся целые типы (SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64), типы с плавающей точкой (Single, Double, Decimal), логический (булев) тип (Boolean) и символьный тип (Char). Базовым классом для всех типов по значению является класс System.ValueType, который наследуется от System.Object (см. п. 3.1.5). Новых методов этот класс не объявляет, перегружая лишь некоторые методы базового класса (в т.ч. Equals и ToString). Однако, все производные от System.ValueType классы, представляющие собой типы данных по значению, реализуют интерфейсы IComparable,

IFormattable, IEquatable и IConvertible (см. п. 4.9.4), поэтому обладают некоторыми общими членами. Также в эти классы добавлены новые члены, присущие этим типам данных (табл. 3.3).

Табл. 3.3 – Общие члены типов данных по значению

Член	Владелец(1)	Описание
Методы		
int CompareTo(<тип> value)	IComparable<T>	Сравнивает данный экземпляр A с заданным значением B (результат отрицательный, если A < B, ноль, если A = B и положительный, если A > B)
int CompareTo(**object** value)	IComparable	То же, но без типизации
bool Equals(<тип> obj)	IEquatable<T>	Возвращает истину, если данных экземпляр равен заданному значению
override bool Equals(**object** obj)	Object	То же, но без типизации
Type GetType()	Object	Возвращает метакласс типа
TypeCode GetTypeCode()	IConvertible	Возвращает TypeCode для типа(2)
static <тип> Parse(**string** s, ...)		Ряд методов для преобразования строки в значение данного типа
override string ToString()	Object	Обратное преобразование – из данного типа в строку
string ToString (IFormatProvider provider)	IConvertible	То же, но с дополнительными опциями (региональные параметры и т.д.)
string ToString(**string** format)		То же, но в указанном формате
string ToString(**string** format, IFormatProvider provider)	IFormattable	То же, но одновременно с форматом и дополнительными опциями

static bool TryParse(**string** s, ..., **out** <тип> result)		Преобразование из строки в значение данного типа с флагом успешного выполнения
<тип> To<класс> (IFormatProvider provider)	IConvertible	Преобразование к указанному типу данных(3)
Поля		
const <тип> MaxValue		Наибольшее возможное значение типа(4)
const <тип> MinValue		Наименьшее возможное значение типа(4)
static readonly string FalseString		Представляет логическое значение **false** в виде строки(5)
static readonly string TrueString		Представляет логическое значение **true** в виде строки(5)

Примечания:

(1) Здесь и далее владелец – это класс, от которого данный класс наследует описываемый член, или интерфейс, член которого реализует данный класс;

(2) Перечисление TypeCode содержит коды типов объектов (табл. 3.4);

(3) Здесь <тип> – это тип данных, а <класс> – имя соответствующего ему класса .NET (например, класс Int16 – тип **short**);

(4) Отсутствует в классах System.Boolean и System.Enum;

(5) Только в классе System.Boolean.

Здесь и далее, если не оговорено другое, рассматриваются только открытые (**public**) члены классов, т.к. только к ним мы имеем доступ.

Табл. 3.4 – Символические константы перечисления TypeCode

Имя	Значение	Описание
Empty	0	Пустая ссылка
Object	1	Универсальный тип для представления любых типов значений и ссылочных типов, которые не могут быть представлены никаким другим значением TypeCode
DBNull	2	Значение **null** в базе данных

`Boolean`	3	Тип **bool** (System.Boolean)
`Char`	4	Тип **char** (System.Char)
`SByte`	5	Тип **sbyte** (System.SByte)
`Byte`	6	Тип **byte** (System.Byte)
`Int16`	7	Тип **short** (System.Int16)
`UInt16`	8	Тип **ushort** (System.UInt16)
`Int32`	9	Тип **int** (System.Int32)
`UInt32`	10	Тип **uint** (System.UInt32)
`Int64`	11	Тип **long** (System.Int64)
`UInt64`	12	Тип **ulong** (System.UInt64)
`Single`	13	Тип **float** (System.Single)
`Double`	14	Тип **double** (System.Double)
`Decimal`	15	Тип **decimal** (System.Decimal)
`DateTime`	16	Тип для представления значений даты и времени (System.DateTime), см. п. 3.1.2.6
`String`	18	Тип **string** (System.String), см. п. 3.1.3.1

3.1.2.1. Целые типы

Список целых типов языка C# приведен в таблице 3.5.

Табл. 3.5 – Целые типы

Тип	Класс	Описание	Диапазон
sbyte	`System.SByte`	8 бит со знаком	–128…127 ($–2^7…2^7–1$)
short	`System.Int16`	16 бит со знаком	–32 768…32 767 ($–2^{15}…2^{15}–1$)
int	`System.Int32`	32 бита со знаком	–2 147 483 648…2 147 483 647 ($–2^{31}…2^{31}–1$)
long	`System.Int64`	64 бита со знаком	–9 223 372 036 854 775 808… 9 223 372 036 854 775 807 ($–2^{63}…2^{63}–1$)
byte	`System.Byte`	8 бит без знака	0…255 ($0…2^8–1$)
ushort	`System.UInt16`	16 бит без знака	0…65 535 ($0…2^{16}–1$)

uint	System.UInt32	32 бита без знака	0…4 294 967 295 (0…2^{32}–1)
ulong	System.UInt64	64 бита без знака	0…18 446 744 073 709 551 615 (0..2^{64}–1)

Как видно, каждый класс, описывающий тип, имеет более короткий псевдоним. Так, например, для описания 16-битного целого числа без знака, можно использовать тип **ushort**, UInt16 или System.UInt16.

Примеры описания целых переменных:

```
System.Int32 x;
int y = 5;
ulong z = 0xfa12;
int n = new Int32(); // 0
```

В первой строке объявлена переменная x, являющаяся 32-битным целым со знаком. Она не инициализирована, поэтому попытка вызвать какие-либо методы класса System.Int32 или попытка присвоить x другой переменной вызовет ошибку компиляции:

```
n = x; // ошибка – использование локальной переменной,
       // которой не присвоено значение
Console.WriteLine(x.ToString()); // то же самое
```

Во второй строке объявляется переменная y такого же типа. Для ее инициализации используется константа 5. Переменная z, 64-битное целое без знака, инициализирована шестнадцатеричной константой. Как и в языке C++, такие константы начинаются с префикса «0x» или «0X». Переменная n инициализирована конструктором по умолчанию. Это эквивалентно инициализации нулем. Следующие варианты инициализации приводят к одинаковому результату:

```
int n1 = new System.Int32();
int n2 = new Int32();
int n3 = new int();
int n4 = 0;
```

Как и в C++, любая целая константа относится к типу **int**. При необходимости она неявно преобразуется к требуемому типу. Если необходимо явно указать тип целой константы, используются суффиксы:

```
123 – int
123U – uint
123L – long
123UL – ulong
```

Регистр суффиксов не важен, буквы U и L для типа **ulong** можно пере-

ставлять местами.

Однако, компилятор принимает во внимание значение самой константы. Так, константа 4000000000 будет иметь тип **uint**, т.к. ее значение выходит за пределы типа **int**. Если значение будет выходить за пределы типа **uint**, будет использован тип **long**, а если и его диапазона не хватит – **ulong**.

3.1.2.2. Типы с плавающей точкой

Список типов с плавающей точкой языка C# приведен в таблице 3.6.

Табл. 3.6 – Типы с плавающей точкой

Тип	Класс	Описание	Диапазон
float	System.Single	32 бита 7 значащих цифр	$\pm 1.5 \cdot 10^{-45} \ldots \pm 3.4 \cdot 10^{38}$
double	System.Double	64 бита 15-16 значащих цифр	$\pm 5.0 \cdot 10^{-324} \ldots \pm 1.7 \cdot 10^{308}$
decimal	System.Decimal	128 бит 28 значащих цифр	$\pm 1.0 \cdot 10^{-28} \ldots \pm 7.9 \cdot 10^{28}$

Все, сказанное выше для целых типов данных, верно и для типов с плавающей точкой. Примеры описания переменных:

```
System.Single s = 12.3f;
double d = 12.3;
decimal m = 12.3m;
```

Опять же, как и в языке C++, константа с плавающей точкой по умолчанию имеет тип **double**. Для указания другого типа данных используются суффиксы:

```
12.3 – double
12.3F, 12.3f – float
12.3M, 12.3m – decimal
```

Для констант типа **double** тоже можно использовать суффикс (d или D), но это не обязательно. Так делают, чтобы показать, что целочисленная константа должна рассматриваться как константа с плавающей точкой. Однако, того же можно добиться, добавив ей ноль после десятичного разделителя:

```
Console.WriteLine(1 / 2);   // 0
Console.WriteLine(1 / 2d);  // 0.5
Console.WriteLine(1 / 2.0); // 0.5
```

Если в приведенном выше примере описания переменных суффиксы не указать, то произойдет ошибка компиляции. Подробнее про явное и неявное

преобразование типов данных мы поговорим в п. 3.1.2.8.

Следует отметить, что для переменных с плавающей точкой доступен не весь диапазон значений – накладываются ограничения, связанные с количеством значащих цифр. Например, в переменной типа **float** может храниться значение 7,123456f (7 значащих цифр), а значение 7,1234567f (8 значащих цифр) – уже нет, одна цифра будет потеряна.

В типе **decimal** диапазон значений был уменьшен для увеличения количества значащих цифр. Т.е. больше бит отводится для мантиссы числа и меньше – для экспоненты. Чаще всего этот тип данных применяется для хранения денежных сумм.

Как известно, все числа хранятся в двоичной системе. Поэтому без погрешностей в памяти может храниться только такое нецелое число, которое является суммой степеней числа 2. Например, число 2,75 может быть сохранено без погрешности:

$$2{,}75 = 2^1 + 2^{-1} + 2^{-2},$$

а число 7,1 – нет. Убедимся в этом:

```
float f = 7.1f;
double d = f;
Console.WriteLine(f);
Console.WriteLine(d);
```

При запуске этого кода на экране отобразятся числа:

```
7,1
7,09999990463257
```

Переменная d имеет больше значащих цифр, поэтому показывает, что на самом деле в переменной f хранится не число 7,1, а некое близкое к нему число. Число 2,75 в обоих случаях отобразилось бы одинаково.

Обратите внимание – при наборе текста программы разделителем целой и дробной части числа является точка, а при выводе на экран – символ, указанный в региональных настройках ОС. Подробнее об этом мы поговорим в § 3.2.

3.1.2.3. Логический тип

Логический тип данных представлен единственным типом (табл. 3.7).

Табл. 3.7 – Логический (булев) тип

Тип	Класс	Описание	Диапазон

bool	System.Boolean	8 бит, булево значение	**false**, **true**

В языке C++ многие типы данных могут быть неявно преобразованы к логическому типу. Например:

```
if (x)
{
      // блок операторов
}
```

Данный код будет успешно компилироваться, если типом переменной будет **bool**, а также, если это будет указатель (на данные, на функцию или процедуру, на метод класса, класс, интерфейс – неважно), целый тип данных (языке C++ символьный тип также считается целым) или тип данных с плавающей точкой. Если значение переменной x равно 0 (NULL для указателей), то оно неявно преобразуется в значение **false**, во всех остальных случаях – в значение **true**. Однако, в языке C# более строгая типизация – данный код будет компилироваться только в том случае, если переменная x имеет тип **bool**:

```
int x;

if (x) // ошибка! неявное преобразование типа "int" в "bool"
       // невозможно; явное тоже: при использовании (bool)x
       // будет показана та же ошибка
{
      // блок операторов
}
```

Для явного преобразования типов могут использоваться операции отношения либо специальные методы:

```
if (x != 0) // или if (!x.Equals(0))
{
      // блок операторов
}
```

3.1.2.4. Символьный тип

Символьный тип данных также представлен единственным типом (табл. 3.8).

Табл. 3.8 – Символьный тип

Тип	Класс	Описание	Диапазон
char	System.Char	16 бит, символ Unicode	U+0000…U+FFFF

Символьные типы со знаком и без знака, т.е. **signed char** и **unsigned char**, имевшиеся в языке C++, в языке C# отсутствуют. Все символы пред-

ставляются как символы Unicode (UTF-16), представляющие собой двухбайтовые значения. Посмотреть, какой код соответствует каждому символу, можно в Таблице символов Windows (рис. 3.1).

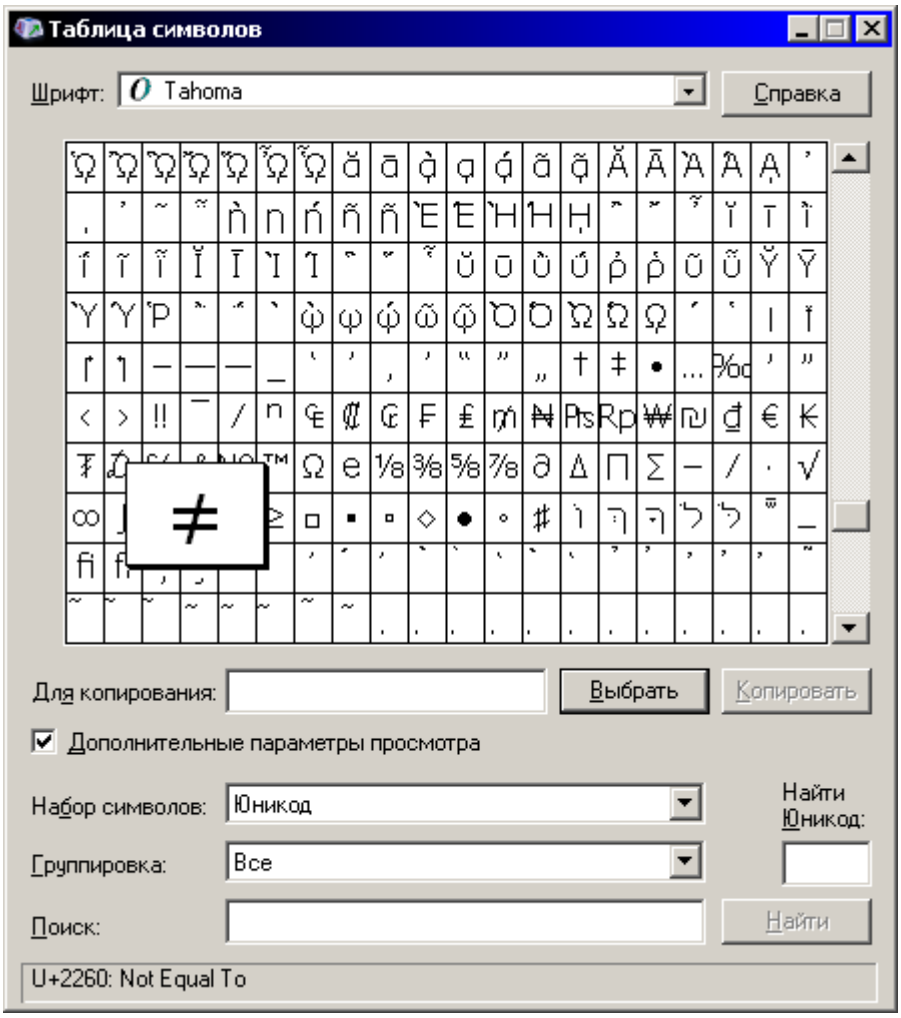


Рис. 3.1 – Таблица символов Unicode

Открыть ее можно через меню «Пуск → Программы → Стандартные → Служебные → Таблица символов», или через меню «Пуск → Выполнить» (Win+R), набрав в нем «charmap». Далее следует в выпадающем списке наборов символов выбрать категорию Unicode.

Для инициализации переменных символьного типа можно использовать символьные литералы (как и в языке C++, заключенные в одинарные кавычки) или явное преобразование целых чисел к символьному типу:

```
System.Char a = 'A';
char b = (char)65;
char c = '\u0041';
char d = '\x0041';
```

В символьных литералах также можно указывать код символа в шест-

надцатеричном виде с префиксом \x или \u. Все описанные переменные будут содержать большую латинскую букву A.

В настоящее время, кроме кодировки UTF-16, используются также UTF-8 и UTF-32. Первая из них позволяет уменьшить размер текста в байтах, если используются в основном символы из начала кодовой таблицы (U+0000… U+07FF). Использование второй обусловлено тем, что в настоящее время в кодировке Unicode имеется около 100 000 символов. Очевидно, что 16 бит для их представления недостаточно, поэтому был разработан стандарт UTF-32. При необходимости некоторые символы Unicode могут представляться последовательностью из двух значений типа **char** (т.е. 4 байтами). Кодировка UTF-8 может, теоретически, использовать до 6 байт для представления символа Unicode (см. ru.wikipedia.org/wiki, статья «Unicode»).

Также есть ряд специальных символов, впрочем, знакомых еще по C++ (табл. 3.9).

Табл. 3.9 – Специальные символы

Символ	Описание	Код символа
\'	Одинарная кавычка	U+0027
\"	Двойная кавычка	U+0022
\0	Конец строки	U+0000
\a	Звонок	U+0007
\b	Возврат (забой)	U+0008
\f	Следующая страница	U+000C
\n	Переход на следующую строку	U+000A
\r	Возврат каретки	U+000D
\t	Горизонтальная табуляция	U+0009
\v	Вертикальная табуляция	U+000B
\\	Обратная косая черта	U+005C

Полный список управляющих символов можно посмотреть на сайте Википедии (ru.wikipedia.org/wiki) в статье «Управляющие символы».

Помимо тех членов, что описаны в п. 3.1.2, класс Char имеет несколько других полезных членов (табл. 3.10).

Табл. 3.10 – Специфические члены класса Char

Член	Описание
`static string`	Преобразует заданный символ Unicode

`ConvertFromUtf32(int utf32)`	из кодировки UTF-32 в UTF-16
static int `ConvertToUtf32(...)`	Обратное преобразование
static double `GetNumericValue(...)`	Преобразует числовой символ Unicode в число типа **double**
static `UnicodeCategory GetUnicodeCategory(...)`	Относит указанный символ Unicode к категории, определенной одним из значений UnicodeCategory(1)
static bool `Is<категория>(...)`	Показывает, относится ли указанный символ Unicode к некоторой категории(2)
static char `ToLower(char c, ...)`	Преобразует значение символа в его эквивалент в нижнем регистре
static char `ToLowerInvariant(char c)`	То же самое, но инвариантно к региональным параметрам
static char `ToUpper(char c, ...)`	Преобразует значение символа в его эквивалент в верхнем регистре
static char `ToUpperInvariant(char c)`	То же самое, но инвариантно к региональным параметрам

Примечания:

(1) При просмотре Таблицы символов Windows тоже можно выбирать одну из категорий символов (см. поле «Группировка» на рис. 3.1). В перечислении UnicodeCategory описано весьма много констант, поэтому мы не будем приводить их список. Посмотреть их можно в справочной системе MSDN.

(2) Речь идет о некоторых категориях, описанных в перечисляемом типе UnicodeCategory. Например, если имеется константа UnicodeCategory.Control, то существует и соответствующий ей метод IsControl. Данных методов также весьма много, полный список можно увидеть в справочной системе MSDN.

3.1.2.5. Перечисляемый тип

В отличие от рассмотренных выше типов данных, при работе с перечисляемым типом (или перечислением, от англ. enumeration) мы не просто используем уже реализованный тип, а создаем свой собственный. Причем, в отличие от языка C++, новый тип нельзя описывать внутри метода. Он должен быть объявлен в каком-либо классе или пространстве имен.

Таким образом, синтаксис определения перечисляемого типа таков:

```
[<атрибуты>] [<модификаторы>] enum <имя перечисления> [: <базовый тип>]
"{"
      <константа1> [= <значение1>] [, <константа2> [= <значение2>]]
            [, ...]
"}" [;]
```

Разберем аспекты описания перечисления:

1) Атрибуты, модификаторы. Т.к. перечисляемый тип является членом класса или пространства имен, при его объявлении можно использовать различные атрибуты и модификаторы, которые используются при объявлении членов классов и пространств имен в целом. Об атрибутах подробнее мы поговорим в § 5.4, а о модификаторах – в § 4.2.

2) Имя перечисления является идентификатором. Об идентификаторах мы будем говорить в п. 3.1.6.

3) Базовый тип. По умолчанию, константы перечисления имеют тип **int**. Но можно задать базовый тип и явным образом, выбрав его среди целочисленных типов данных. Необходимость в этом возникает, когда значения констант невозможно или неудобно представлять с помощью типа **int**. Пример:

```
enum DayOfWeek : byte { }
```

4) Константы. Для каждой константы задается ее символическое имя (по сути, являющееся идентификатором-константой). Имена перечисляемых констант внутри каждого перечисления должны быть уникальными. Пример:

```
enum DayOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
                 Saturday }
```

5) Значения. По умолчанию константам присваиваются последовательные значения базового типа, начиная с 0. Но можно задать и собственные значения (причем они могут и совпадать для различных констант), например:

```
enum DayOfWeek { Sunday = 1, Monday = 2, Tuesday = 3, Wednesday = 4,
                 Thursday = 5, Friday = 6, Saturday = 7 }
```

Если значение не указано, оно вычисляется прибавлением единицы к значению предыдущей константы. Пример:

```
enum Nums { two = 2, three, four, ten = 10, eleven, fifty = ten + 40 }
```

Константам three и four присваиваются значения 3 и 4, константе eleven – 11, fifty – 50.

Преимущество перечисления перед описанием именованных констант состоит в том, что связанные константы нагляднее. Кроме того, компилятор

выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список.

Перечисления удобно использовать для представления битовых флагов, например:

```
enum BitFlags : byte
{
      bit1 = 0x01, bit2 = 0x02, bit3 = 0x04, bit4 = 0x08,
      bit5 = 0x10, bit6 = 0x20, bit7 = 0x40, bit8 = 0x80
}
```

Все перечисления в языке C# являются потомками базового класса System.Enum (см. п. 3.1.5), от которого они наследуют некоторые полезные методы:

```
static string GetName(Type enumType, object value);
static string[] GetNames(Type enumType);
static Array GetValues(Type enumType);
static bool IsDefined(Type enumType, object value);
static Type GetUnderlyingType(Type enumType);
static object ToObject(Type enumType, <тип> value);
```

И, конечно, сюда входят все члены, описанные в п. 3.1.2. Пример использования некоторых методов класса Enum:

```
enum DayOfWeek { Monday = 1, Tuesday, Wednesday, Thursday,
      Friday, Saturday, Sunday } // 1
[FlagsAttribute] enum BitFlags : byte { bit1 = 0x01,
      bit2 = 0x02, bit3 = 0x04, bit4 = 0x08, bit5 = 0x10,
      bit6 = 0x20, bit7 = 0x40, bit8 = 0x80 } // 2

static int Main(string[] args)
{
      DayOfWeek dow = DayOfWeek.Friday; // 3
      Console.WriteLine(Enum.GetName(dow.GetType(), dow)); // 4
      Console.WriteLine(Enum.GetName(typeof(BitFlags), 8)); // 5

      BitFlags bits = BitFlags.bit2 | BitFlags.bit4; // 6
      Console.WriteLine(bits); // 7

      string[] names = Enum.GetNames(typeof(DayOfWeek)); // 8
      Console.Write("В перечислении DayOfWeek " + names.Length + "
элементов: "); // 9
      foreach (string name in names) Console.Write(name + " "); // 10

      Array values = Enum.GetValues(typeof(BitFlags)); // 11
      Console.Write("\nЗначения констант перечисления BitFlags: ");
      foreach (byte value in values) Console.Write(value + " "); // 12

      Console.WriteLine("\nВ перечислении DayOfWeek константа Tuesday "
+ (Enum.IsDefined(typeof(DayOfWeek), "Tuesday") ? "\" присутствует" : "\"
отсутствует")); // 13
      Console.WriteLine("В перечислении BitFlags константа со значением
\"" + 7 + (Enum.IsDefined(typeof(BitFlags), (byte)7) ? "\" присутствует" :
"\" отсутствует")); // 14
```

```
            Console.WriteLine("Тип элементов в перечислении BitFlags: " +
Enum.GetUnderlyingType(typeof(BitFlags))); // 15

            TypeCode code = dow.GetTypeCode(); // 16
            Console.WriteLine("Тип элементов в перечислении DayOfWeek: " +
code); // 17

            BitFlags bits2 = (BitFlags)Enum.ToObject(typeof(BitFlags), 0x55);
// 18
            Console.WriteLine(bits2); // 19
            return 0;
      }
```

При запуске программа выведет на консоль следующие данные:

```
      Friday
      bit4
      bit2, bit4
      В перечислении DayOfWeek 7 элементов: Monday Tuesday Wednesday Thursday
Friday Saturday Sunday
      Значения констант перечисления BitFlags: 1 2 4 8 16 32 64 128
      В перечислении DayOfWeek константа "Tuesday" присутствует
      В перечислении BitFlags константа со значением "7" отсутствует
      Тип элементов в перечислении BitFlags: System.Byte
      Тип элементов в перечислении DayOfWeek: Int32
      bit1, bit3, bit5, bit7
```

 Пример: Samples\3.1\3_1_2_enum.

В данном примере важные для нас строки маркированы комментариями. Разберем помеченные их:

1) Описываем перечисление, содержащее дни недели. Значения констант начинаются с 1.

2) Описываем перечисление, содержащее маски для различных битов в байте: bit1 для младшего бита, bit8 – для старшего. Тип элементов перечисления – **byte**. Атрибут FlagsAttribute указывает, что перечисление может обрабатываться как битовое поле, которое является набором флагов. Про атрибуты мы будем говорить в § 5.4.

3) Пример описания переменной типа DayOfWeek и инициализация ее значением DayOfWeek.Friday. В общем случае, синтаксис доступа к именованной константе перечисляемого типа таков:

```
      [<пространство имен>][<класс>]<тип перечисления>.<константа>
```

Но если тип описан в тех же пространстве имен и классе, в которых используется, то указывать их не обязательно. В принципе, вместо записи DayOfWeek можно использовать запись Program.DayOfWeek. Но, чтобы не усложнять программу ненужными конструкциями, так делается только в том случае, если перечисление объявлено в другом классе. А если бы наш класс

был включен в пространство имен Sample_3_1_2_enum, то можно было бы использовать запись Sample_3_1_2_enum.Program.DayOfWeek. Опять же, так следует делать только в том случае, если перечисление используется в другом пространстве имен (более того, только если возможен конфликт имен, т.к. после использования директивы using Sample_3_1_2_enum указывать данное пространство имен при доступе к его членам не обязательно).

4) Статический метод GetName позволяет получить символическое имя константы по ее номеру. Первый параметр метода – метакласс, или объект, представляющий собой тип перечисления. Подробнее о метаклассах мы будем говорить в § 5.3. Получить метакласс для любого типа можно при помощи оператора языка C# **typeof** (о нем также поговорим позже, в § 3.3), либо, для перечислений, при помощи метода System.Enum.GetType. В данном случае используется второй вариант. Метод System.Enum.GetType не является статическим, поэтому для его вызова необходимо использовать экземпляр перечисления (в данном случае – dow). Второй параметр метода GetName – числовое значение константы. Здесь можно использовать dow, DayOfWeek.Friday или 5 – результат будет одинаковым, на консоли отобразится строка «Friday».

5) В данном случае для получения метакласса перечисления BitFlags был использован оператор **typeof**. Для этого создание экземпляра перечисления не требуется. Во втором параметре метода GetName можно указать 8 или BitFlags.bit4, в результате на консоль будет выведена строка «bit4».

6) Т.к. перечисление BitFlags содержит битовые поля, при конструировании экземпляра данного перечисления можно комбинировать несколько констант, используя операцию побитовой дизъюнкции (подробнее об операциях в § 3.3).

7) Как мы уже говорили, многие объекты перегружают виртуальный метод ToString, который наследуют от базового класса System.Object. Перегружает его и класс System.Enum. Для экземпляра переменной перечисляемого типа при выводе на консоль выводится значение этой переменной. Например, в строке 4 получить аналогичный результат мы могли еще проще:

```
    Console.WriteLine(dow);
```

В данном случае на консоли мы увидим строку «bit2, bit4». Если бы атрибут FlagsAttribute не был указан, компилятор не знал бы, что значение bits может являться комбинацией констант перечисления, и вывел бы на экран

просто число «10» (bit2 | bit4 = $0x02_{16}$ | $0x08_{16}$ = $0x0a_{16}$ = 10_{10}), т.к. константы со значением 10 в перечислении нет.

8) Статический метод GetNames формирует массив имен констант, составляющих перечисление. О массивах мы будем говорить позже, при изучении ссылочных типов данных. В качестве параметра указываем метакласс перечисления.

9) Выводим на экран размер массива names. На консоль будет выведена строка «В перечислении DayOfWeek 7 элементов».

10) В цикле foreach (см. § 3.4) перебираем все элементы массива names и выводим их на консоль:

```
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

11) Статический метод GetValues формирует массив значений констант, составляющих перечисление.

12) Выводим на экран значения констант:

```
1 2 4 8 16 32 64 128
```

13) Статический метод IsDefined возвращает значение **true**, если константа с заданным символическим именем описана в указанном перечислении, и **false** в противном случае. У данного метода два аргумента. Первый – это метакласс перечисления. Второй – искомая символическая константа. Причем ее можно задавать в любом формате. В данном случае используется строковое представление. Оператор «?:» по логике своей работы соответствует тренарному оператору языка C++. Т.к. константа «Tuesday» в перечислении DayOfWeek определена, видим на консоли строку «В перечислении DayOfWeek константа "Tuesday" присутствует».

14) Здесь используется числовое представление константы во втором аргументе метода IsDefined. Т.к. константы со значением 7 в перечислении BitFlags нет, видим на консоли строку «В перечислении BitFlags константа со значением "7" отсутствует».

15) Статический метод GetUnderlyingType возвращает имя базового типа, на котором построено перечисление. Возвращает он метакласс базового типа, а в его строковое представление при выводе на консоль формируется вызовом перегруженного метода ToString. Для метакласса он возвращает имя класса-оригинала. Для перечисления BitFlags этот метод вернет **typeof**(**byte**), поэтому видим на экране базовый тип «System.Byte».

16) Метод GetTypeCode тоже возвращает информацию о базовом типе перечисления, но уже в виде экземпляра перечисления TypeCode.

17) Здесь мы снова пользуемся тем же фактом, что и в строке 7: любой объект может быть представлен в виде строки. На консоль выводится текст «Тип элементов в перечислении DayOfWeek: Int32».

18) Метод ToObject имеет несколько реализаций, отличающихся типом второго аргумента (есть реализации для всех целых типов и типа **object**). В данном случае использовалась целая константа, поэтому вызывался метод

```
public static object ToObject(Type enumType, int value);
```

Данный метод позволяет преобразовать значения, представленные в виде ряда типов данных в значение перечисляемого типа. Первый аргумент – метакласс перечисления, второй – преобразуемое значение. Метод возвращает результат в виде ссылки на **object**, поэтому требуется преобразование типа к BitFlags.

19) Константа $0x55_{16} = 01010101_2$, поэтому в перечисление были включены символические константы bit1, bit3, bit5 и bit7, в чем мы и убеждаемся, выводя bits2 на консоль.

В целом, для тех, кто только начал изучение языка C#, в данном примере много неизвестных конструкций, но все они будут рассмотрены ниже.

3.1.2.6. Структуры

Структуры – еще один вид определяемых пользователем типов данных. Структуры наследуются напрямую от класса System.ValueType, поэтому не реализуют тех интерфейсов и их методов, которые перечислены в п. 3.1.2. Структуры обычно используются для инкапсуляции небольших объектов, в противном случае лучше написать класс.

Синтаксис описания структур следующий:

```
[<атрибуты>] [<модификаторы>] struct <имя структуры>
      [: <интерфейс1> [, <интерфейс2>] [, ...]]
"{"
      [<тело структуры>]
"}" [;]
```

Аналогично перечислениям, объявлять структуры в теле методов нельзя. Атрибуты и модификаторы имеют тот же смысл, что и при описании перечисляемого типа. Их мы рассмотрим позже. В теле структуры могут быть объявлены любые члены – конструкторы, поля, свойства, события и т.д. Все

это будет рассматриваться в главе 4 «Классы и интерфейсы». Новый экземпляр структуры, как и любого типа по значению, можно создать, используя оператор «=» (выражение в правой части должно являться структурой такого же типа) или оператор **new**:

```
<экземпляр> = <выражение>;
<экземпляр> = new <имя структуры> ([<параметры конструктора>]);
<экземпляр> = new <имя структуры> "{" [<поле> = <значение> [, ...]] "}"
```

Если рассматривать язык C++, то можно прийти к выводу, что в нем классы и структуры отличаются немногим: члены структур по умолчанию имеют модификатор доступа **public**, а члены классов – **private**. В языке C# различия между классами и структурами более значительны:

1) Структура не может наследоваться от другой структуры или класса и не может быть основой для других классов. Структуры могут реализовывать только интерфейсы (см. § 4.9). Пример:

```
interface ITest { }
class CTest { }
struct MyStruct1 : CTest, ITest { } // Ошибка
struct MyStruct2 : ITest { } // OK
class MyClass1 : CTest, ITest { } // OK
class MyClass2 : MyStruct2 { } // Ошибка
```

2) В объявлении структуры поля не могут быть инициализированы до тех пор, пока они не будут объявлены как постоянные или статические. Пример:

```
struct MyStruct
{
      public int A1 = 1; // Ошибка
      public const int A2 = 1; // OK
      public int A3; // OK
}

class MyClass
{
      public int A1 = 1; // OK
      public const int A2 = 1; // OK
}
```

Постоянные и статические члены классов рассмотрены в главе 4.

3) Структура не может объявлять используемый по умолчанию конструктор (конструктор без параметров) и деструктор. Пример:

```
struct MyStruct2
{
      public int A3;
      public MyStruct2(int a) { A3 = a; } // OK
}
```

```
struct MyStruct3
{
      public MyStruct3() { } // Ошибка
      ~MyStruct3() { } // Ошибка
}

class MyClass1
{
      public int A1;
      public MyClass1(int a) { A1 = a; } // OK
      ~MyClass1() { } // OK
}

class MyClass3
{
      public MyClass3() { }
}

class MyClass4
{
      public int a;
}

static int Main(string[] args)
{
      MyStruct2 x1 = new MyStruct2(); // OK
      MyStruct2 x2 = new MyStruct2(1); // OK
      MyClass1 y1 = new MyClass1(); // Ошибка
      MyClass1 y2 = new MyClass1(1); // OK
      MyClass4 y4 = new MyClass4(); // OK
      return 0;
}
```

В структуре MyStruct3 была сделана попытка объявления конструктора по умолчанию и деструктора. Результат – ошибка компиляции. В классе MyClass3 есть конструктор по умолчанию, а в классе MyClass1 – деструктор, и это ошибкой не является. Однако, при создании экземпляра структуры (x1) мы можем использовать конструктор по умолчанию. Он добавляется компилятором неявно и инициализирует все поля структуры значениями по умолчанию.

При создании класса конструктор по умолчанию неявно добавляется только в том случае, если других конструкторов в классе нет (y4). Если же в классе описаны только конструкторы с параметрами, использовать при создании экземпляров конструктор по умолчанию нельзя (y1).

И, естественно, можем также использовать явно описанные конструкторы с параметрами структур (x2) и любые явно описанные конструкторы классов (y2).

4) В отличие то классов, структуры могут быть созданы без использо-

вания оператора **new**. Пример:

```
struct MyStruct2
{
      public int A3;
}

class MyClass1
{
      public int A1;
}

static int Main(string[] args)
{
      MyStruct2 x5;
      Console.WriteLine(x5.A3); // Ошибка
      x5.A3 = 1; // OK
      Console.WriteLine(x5.A3); // OK

      MyClass1 y5;
      y5.A1 = 1; // Ошибка
      y5 = new MyClass1();
      y5.A1 = 1; // OK
      return 0;
}
```

При описании переменной x5 был создан экземпляр структуры MyStruct2. Но, т.к. никакой конструктор не вызывался, ее поля остались неинициализированными. Поэтому при попытке использования любого поля этой структуры на чтение мы видим ошибку компилятора. После того, как поле A3 инициализировано (вызовом конструктора класса **int**, присвоением числовой константы или другой переменной), мы можем его использовать в операторе вывода. На консоли появится число 1. При описании переменной y5 экземпляр класса MyClass1 не создается! Поэтому любая попытка использовать поля этого экземпляра влечет ошибку компиляции.

Также при создании структуры можно сразу инициализировать некоторые поля:

```
MyStruct2 x51;
x51 = new MyStruct2 { A3 = 1, B = 0.5, C = "!!!" };
```

5) Самым серьезным отличием, как уже было сказано, является то, что структуры – это типы по значению, а классы – ссылочные типы. Поэтому при присваивании структуры копируют свое содержимое, а классы – только ссылку. Вспомним, когда в языке C++ происходит копирование значения одной переменной в другую переменную:

• при использовании оператора присваивания;

• при передаче переменной в качестве аргумента по значению (напом-

ним, что кроме как по значению, в языке C++ можно также передавать аргументы по ссылке и по указателю) в процедуру, функцию или метод класса;

• при возврате по значению результата функции или метода класса.

В языке C# во всех этих случаях копирование, как уже было сказано, осуществляется только для структур, классы копируют только ссылку. Передачу аргументов и возврат значений мы разберем в описании методов класса в § 4.4. Следующий пример поясняет различия при использовании оператора присваивания:

```
struct MyStruct2
{
      public string C;
      public MyStruct2(string c)
      {
            C = c;
      }
}

class MyClass1
{
      public string C;
      public MyClass1(string c)
      {
            C = c;
      }
}

static int Main(string[] args)
{
      MyStruct2 x2 = new MyStruct2("mystruct");
      MyClass1 y2 = new MyClass1("myclass");
      MyStruct2 x6 = x2;
      MyClass1 y6 = y2;
      x6.C = "изменили x6.C";
      y6.C = "изменили y6.C";
      Console.WriteLine(x2.C); // mystruct
      Console.WriteLine(y2.C); // изменили y6.C
      return 0;
}
```

Итак, первый вызов метода Console.WriteLine выведет строку «mystruct», потому что мы скопировали содержимое экземпляра структуры x2 в экземпляр x6, и модификация полей структуры x6 никоим образом не влияет на содержимое структуры x2. Второй вызов выведет строку «изменили y6.C», потому что y6 – это ссылка на тот же экземпляр класса, на который ссылается y2. (рис. 3.2). На самом деле, картина не совсем такая, т.к. строки – это ссылочные типы (см. пункт 3.1.3) и, в свою очередь, размещают данные в управляемой динамической куче, но в целом данный рисунок отражает ситуацию.

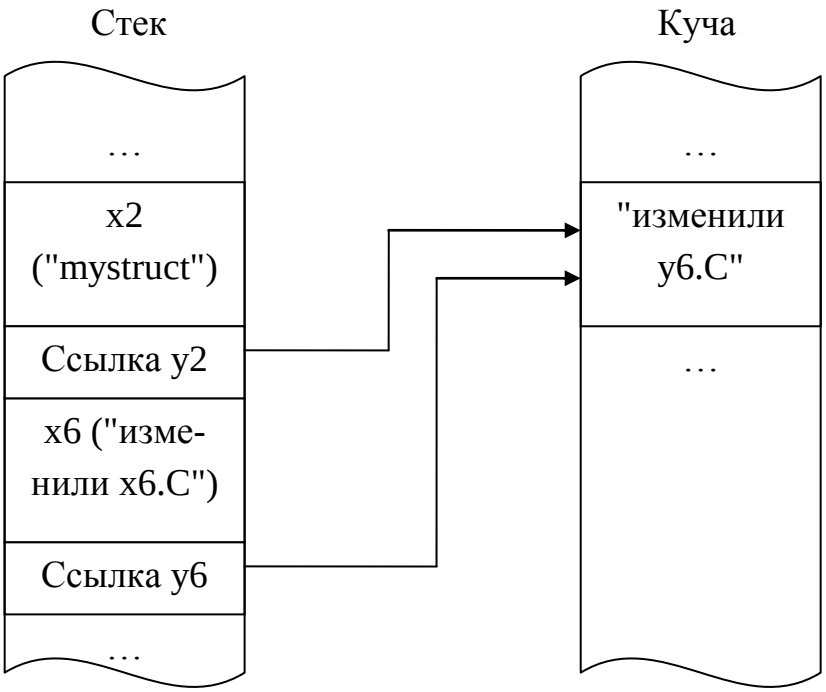


Рис. 3.2 – Содержимое памяти после копирования класса и структуры

Пример: Samples\3.1\3_1_2_struct.

В библиотеке .NET существует достаточно много предопределенных структур. Особое место среди них занимает структура DateTime. Во-первых, она имеет собственную символическую константу в перечислении TypeCode, рассмотренном выше. Всем остальным структурам соответствует константа Object. Во-вторых, для этой структуры переопределены некоторые операции языка (см. § 3.3). В-третьих, в языке C# предусмотрены богатые возможности форматирования, а также консольного ввода/вывода даты и времени (подробнее см. § 3.2). Основная функциональность данной структуры приведена в приложении А.

3.1.2.7. Обнуляемые типы

Часто возникает необходимость в указании для какой-либо переменной флага «значение отсутствует». Например, стоит задача: найти в массиве наибольшее число, не превышающее некоторого заданного числа n. На языке C++ код может выглядеть так:

```
int arr[10] = { /* инициализация массива */ };
int x = -10000; /* некоторое большое отрицательное число */

for (int i = 0; i < 10; i++)
```

```
{
      if(arr[i] <= n && arr[i] > x) x = arr[i];
}

cout << x << endl;
```

В этом коде есть проблема. Наибольшее число может быть меньше, чем –10000, и тогда мы увидим некорректный результат работы программы. И даже если вместо –10000 использовать константу, являющуюся минимальным значением типа **int**, проблема останется – есть вероятность, что в массиве чисел, не превышающих n, нет вообще. Тогда потребуется введение дополнительного флага:

```
int arr[10] = { /* инициализация массива */ };
int minpos = -2147483648; /* минимальное целое число */
bool find = false;

for (int i = 0; i < 10; i++)
{
      if (arr[i] <= n && arr[i] > x)
      {
            x = arr[i];
            find = true;
      }
}

if(find) cout << x << endl;
else cout << "не найдено!\n";
```

Такое удвоение количества переменных не очень удобно. Поэтому в языке C# введены *обнуляемые типы данных*.

Обнуляемые типы могут представлять все значения базового типа и дополнительное пустое значение. Обнуляемый тип обозначается как `<тип>?`, где `<тип>` – базовый тип (любой ранее рассмотренный тип по значению). Данный синтаксис представляет собой сокращенное обозначение типа структуры System.Nullable<тип>. Данная структура имеет формат универсального типа (см. § 5.1).

Экземпляру обнуляемого типа можно присваивать как значения базового типа, так и значение **null**, означающее отсутствие значения. Члены структуры Nullable<T> перечислены в табл. 3.11.

Табл. 3.11 – Члены структуры System.Nullable<T>

Член	Описание
Конструкторы	
`Nullable(T value)`	Инициализирует новый экземпляр структуры заданным значением

Методы	
`T GetValueOrDefault()`	Извлекает текущее значение, если оно не **null**, иначе значение типа T по умолчанию
`T GetValueOrDefault(T defaultValue)`	Извлекает текущее значение, если оно не **null**, иначе заданное значение по умолчанию
Свойства	
bool `HasValue`	Возвращает значение, указывающее, имеет ли значение текущий объект
`T Value`	Возвращает текущее значение

Экземпляр, свойство HasValue которого имеет значение **true**, считается непустым. Свойство Value непустого экземпляра содержит возвращаемое значение.

Экземпляр, свойство HasValue которого имеет значение **false**, считается пустым. Пустой экземпляр имеет неопределенное значение. При попытке чтения свойства Value пустого экземпляра порождается исключительная ситуация System.InvalidOperationException. Процесс доступа к свойству Value обнуляемого экземпляра называется *развертыванием*.

Помимо конструктора по умолчанию, для каждого обнуляемого типа <тип>? предусмотрен открытый конструктор, принимающий один аргумент типа <тип>. При вызове конструктора со значением «x» типа <тип> в виде

```
new T?(x)
```

создается непустой экземпляр типа <тип>?, свойству Value которого присваивается значение «x». Процесс создания непустого экземпляра обнуляемого типа с использованием заданного значения называется *свертыванием*.

Обнуляемые или ссылочные типы (см. пункт 3.1.3) не могут использоваться в качестве базовых для обнуляемого типа. Например, **int**?? (аналог Nullable<Nullable<**int**>>) и **string**? (Nullable<**string**>) являются недопустимыми типами.

Поиск наименьшего положительного элемента в массиве с использованием обнуляемого типа будет выглядеть так:

```
int[] arr = new int[10] { /* инициализация массива */ };
int? x = null;

for (int i = 0; i < 10; i++)
{
      if (arr[i] <= n && (x == null || arr[i] > x)) x = arr[i];
}
```

```
if (x.HasValue) Console.WriteLine(x);
else Console.WriteLine("не найдено!");
```

Как видим из данного кода, свойство Value использовать не обязательно, компилятор сам выполняет преобразование типа, а записи «x != **null**» и «x.HasValue» эквивалентны.

 Пример: Samples\3.1\3_1_2_nullable.

3.1.2.8. Преобразование типов данных

Преобразование типов данных может быть явным и неявным (табл. 3.11). Простейший синтаксис, обеспечивающий неявное преобразование типов, следующий:

```
objA = objB;
```

В этом случае, если неявное преобразование типа экземпляра objB в тип экземпляра objA существует (в таблице стоит плюс), компилятор выполняет его, иначе выдает ошибку.

Синтаксис явного преобразования типов (звездочки в таблице 3.11):

```
objA = (<тип>)objB;
```

Здесь «тип» – такой тип данных, который совпадает с типом экземпляра objA или из которого существует неявное преобразование в тип экземпляра objA.

Для обнуляемого типа преобразование

```
<тип>? objA = objB;
```

не вызовет ошибки в том случае, если типы экземпляров совпадают или существует неявное преобразование из типа объекта objB в тип <тип>.

Табл. 3.11 – Явные и неявные преобразования типов данных

В / ИЗ	sbyte	byte	short	ushort	int	uint	long	ulong	char	float	double	decimal
sbyte		*	+	*	+	*	+	*	*	+	+	+
byte	*		+	+	+	+	+	+	*	+	+	+
short	*	*		*	+	*	+	*	*	+	+	+
ushort	*	*	*		+	+	+	+	*	+	+	+
int	*	*	*	*		*	+	*	*	+	+	+

uint	*	*	*	*	*		+	+	*	+	+	+
long	*	*	*	*	*	*		*	*	+	+	+
ulong	*	*	*	*	*	*	*		*	+	+	+
char	*	*	*	+	+	+	+	+		+	+	+
float	*	*	*	*	*	*	*	*	*		+	*
double	*	*	*	*	*	*	*	*	*	*		*
decimal	*	*	*	*	*	*	*	*	*	*	*	

Пример:

```
int a = 7;
long b = a; // OK
int c1 = b; // Ошибка
int c2 = (int)b; // OK
int c3 = (byte)b; // OK
```

Итак, в первом случае неявное преобразование из типа **int** в тип **long** выполняется успешно, потому что диапазон типа **long** полностью перекрывает диапазон типа **int**. Обратное преобразование во втором случае влечет ошибку компиляции. В третьем случае происходит явное преобразование из типа **int** в тип **long**, а в четвертом – сначала явное преобразование из **long** в **byte**, а затем – неявное из **byte** в **int**. Если в процессе неявного преобразования значение экземпляра objB выходит за диапазон допустимых значений типа экземпляра objA, происходит переполнение (и, как следствие, потеря данных):

```
long big = 10000000000;
int small = (int)big;
Console.WriteLine(small);
```

В данном случае мы увидим на консоли число 1410065408. Почему? Потому что 10000000000_{10} = $0x2540BE400_{16}$, тип Int32 может вместить только 4 байта (т.е. $0x540BE400_{16}$ = 1410065408_{10}), остальные теряются. Поэтому в таблице и поставлены знаки вопроса – нельзя был полностью уверенным в результате явного преобразования.

Как отследить переполнение? Для этого в языке C# существуют такие операторы, как **checked** и **unchecked**. Первый из них включает проверку переполнения, а второй – отключает (по умолчанию проверка отключена). Если при включенной проверке возникает переполнение, генерируется исключительная ситуация System.OverflowException (подробнее об исключительных ситуациях – в § 3.4). Пример:

```
try
{
      short s1 = 30000;
      byte b1 = checked((byte)s1);
      Console.WriteLine("b1 = " + b1);
}
catch (Exception e)
{
      Console.WriteLine("b1: " + e);
}

checked
{
      try
      {
            short s2 = 30000;
            byte b2 = (byte)s2;
            Console.WriteLine("b2 = " + b2);
      }
      catch (Exception e)
      {
            Console.WriteLine("b2: " + e);
      }
}

unchecked
{
      try
      {
            short s5 = 30000;
            byte b5 = checked((byte)s5);
            Console.WriteLine("b5 = " + b5);
      }
      catch (Exception e)
      {
            Console.WriteLine("b5: " + e);
      }
}

checked
{
      try
      {
            short s6 = 30000;
            byte b6 = unchecked((byte)s6);
            Console.WriteLine("b6 = " + b6);
      }
      catch (Exception e)
      {
            Console.WriteLine("b6: " + e);
      }
}
```

Результат работы данного фрагмента кода:

```
      b1: System.OverflowException: Переполнение в результате выполнения
арифметической операции.
      b2: System.OverflowException: Переполнение в результате выполнения
арифметической операции.
      b5: System.OverflowException: Переполнение в результате выполнения
```

```
арифметической операции.
      b6 = 48
```

Для переменной b1 проверка ошибок была включена только в момент преобразования, а для переменной b2 – для целого блока операторов (варианты синтаксиса операторов **checked** и **unchecked** мы разберем в § 3.3). В обоих случаях будет генерироваться исключительная ситуация, т.к. в момент преобразования проверка включена. Для переменной b5 проверка для блока операторов была отключена, но в момент преобразования ее временно включили. Т.к. приоритет имеет ближайший из операторов **checked**/**unchecked**, опять видим сообщение об исключительной ситуации. Для переменной b6 ситуация обратная – оператор **unchecked** описан ближе, чем **checked**. Т.к. в момент преобразования проверка отключена, сообщение об ошибке не появляется, видим на консоли «48» – усеченное до одного байта значение 30000.

Также для явного преобразования значений в библиотеке .NET предусмотрен класс System.Convert (наследуется от System.Object). Он содержит множество методов для преобразования типов данных. Наиболее часто используемые методы имеют следующий формат:

```
      public static <тип1> To<класс>(<тип2> value);
```

Здесь «тип2» – тип преобразуемого экземпляра value, «тип1» – тип, к которому мы преобразуем значение, а «класс» – имя класса .NET, соответствующее этому типу. Например, в предыдущем примере мы преобразовывали тип **short** к типу **byte**. Соответствующий метод класса Convert будет иметь следующую сигнатуру:

```
      public static byte ToByte(short value);
```

Независимо от того, включена проверка переполнения или нет, при ошибках преобразования типов данных классом Convert будут генерироваться исключительные ситуации:

1) InvalidCastException при попытке преобразования

• типа Char к типам Boolean, Single, Double, Decimal или DateTime или наоборот;

• типа DateTime к любому другому типу, кроме String, и, наоборот, любого типа, кроме String, к типу DateTime. Об этом мы еще поговорим при изучении форматирования в § 3.2.

2) FormatException при попытке преобразования

• строки, не равной Boolean.TrueString или Boolean.FalseString к типу

Boolean;

• строки, содержащей несколько символов, к типу Char;

• недопустимой строки в числовой тип или DateTime.

3) OverflowException при потере данных.

Пример:

```
try
{
      short s3 = 30000;
      byte b3 = Convert.ToByte(s3);
      Console.WriteLine("b3 = " + b3);
}
catch (Exception e)
{
      Console.WriteLine("b3: " + e);
}

unchecked
{
      try
      {
            short s4 = 30000;
            byte b4 = Convert.ToByte(s4);
            Console.WriteLine("b4 = " + b4);
      }
      catch (Exception e)
      {
            Console.WriteLine("b4: " + e);
      }
}
```

Результаты работы:

```
      b3: System.OverflowException: Значение было недопустимо малым или
недопустимо большим для беззнакового байта.
      b4: System.OverflowException: Значение было недопустимо малым или
недопустимо большим для беззнакового байта.
```

Как мы видим, независимо от включения или выключения проверки, класс Convert генерирует исключительную ситуацию при возникновении переполнения.

 Пример: Samples\3.1\3_1_2_types.

3.1.3. Типы данных по ссылке

Итак, все классы, наследуемые от System.ValueType, являются типами данных по значению. Все остальные классы, наследуемые от System.Object – это типы данных по ссылке. Ссылочные типы подразделяются на классы, интерфейсы, массивы и делегаты. Некоторые типы классов имеют особое значение:

• System.Object – первичный базовый класс для всех типов;
• System.String – строковый тип языка C#;
• System.ValueType – базовый класс для всех типов значений;
• System.Enum – базовый класс для всех перечисляемых типов;
• System.Array – базовый класс для всех типов массивов;
• System.Delegate – базовый класс для всех типов делегатов;
• System.Exception – базовый класс для всех типов исключений;
• System.Attribute – базовый класс для всех атрибутов.

Здесь мы рассмотрим только строки и массивы, остальные классы рассматриваются далее.

Вообще, когда речь идет о ссылочных типах, не имеются в виду ссылки, аналогичные ссылкам C++:

```
<тип> &<идентификатор> = <выражение>;
```

В языке C++ ссылка размещается на стеке и содержит только адрес на другой объект (экземпляр класса, встроенного типа данных и т.п.), который может быть размещен как на стеке, так и в динамической куче. Указатель C++ тоже содержит адрес, но в явной форме. Ссылка же маскирует тот факт, что она содержит адрес, позволяя работать с ней, как с обычным экземпляром по значению. Ссылка в C++ обязательно должна быть инициализирована (при попытке объявить неинициализированную ссылку компилятор сразу выдает ошибку), а указатель может не инициализироваться (содержать NULL).

Ссылка в языке C# комбинирует свойства ссылок и указателей C++. Во-первых, сама ссылка хранится на стеке, а данные, на которые она указывает – в управляемой динамической куче. Невозможно получить ссылку на объект, хранящийся на стеке (т.е. на экземпляр типа по значению)! Это легко проверить:

```
int value = 5;
object refobj = value;

value = 7;
Console.WriteLine(value);
Console.WriteLine(refobj);
Console.WriteLine(Object.ReferenceEquals(value, refobj));
```

На консоли видим:

```
7
5
False
```

Если бы ссылка refobj указывала на экземпляр value, то после операции value = 7 ссылка refobj указывала бы на объект со значением 7, а она, как видно, по-прежнему указывает на объект со значением 5. Вызов метода ReferenceEquals подтверждает это. Получается, что в момент операции refobj = value был создан новый ссылочный объект, в который была помещена копия объекта-значения, т.е. произошла упаковка типа (см. пункт 3.1.5).

Во-вторых, ссылка может не быть инициализированной, т.е. содержать значение **null**. До инициализации использовать экземпляр ссылочного типа нельзя, это приведет к ошибке компиляции. Вызов методов экземпляра, инициализированного пустой ссылкой (**null**), приведет к ошибке при выполнении программы. Пример:

```
object obj;
obj.ToString(); // Ошибка компиляции
obj = new Object();
obj.ToString(); // OK
Object obj2 = null;
obj2.ToString(); // Ошибка во время выполнения
```

Синонимом класса System.Object является ключевое слово языка C# **object**.

3.1.3.1. Строки

Строки – очень важный тип данных. В языке C++ нет встроенной поддержки строк, из-за чего у многих начинающих программистов (особенно писавших ранее на языке Pascal) возникали проблемы. Позже строки были реализованы в различных библиотеках (STL, VCL и т.д.) в виде классов.

 Пример: Samples\3.1\3_1_3_string.

Неизменяемые строки

Строки в языке C# представлены классом System.String (синоним – ключевое слово **string**). Строка представлена массивом символов Unicode. Однако, проводить аналогию между классом String и массивом из элементов типа **char** нельзя, хотя существуют способы преобразования между двумя этими способами представления строк.

Как любой элемент программы, заключенный в одинарные кавычки, является экземпляром типа **char**, так и любой элемент программы, заключенный в двойные кавычки, является экземпляром типа **string**.

Основные члены класса String представлены в табл. Б.1 (приложение Б). Пример (ввиду большого количества членов ограничимся демонстрацией только некоторых из них):

```
string s1 = string.Empty; // 1
string s2 = new string("Hello!".ToCharArray()); // 2
string s3 = "Hello!"; // 3
string s4 = s3; // 4

Console.WriteLine("s1 = " + s1);
Console.WriteLine("s2 = " + s2);
Console.WriteLine("s3 = " + s3);
Console.WriteLine("s4 = " + s4);
Console.WriteLine(Object.ReferenceEquals(s3, s4) ?
      "ref s3 == ref s4" : "ref s3 != ref s4"); // 5

s4 = "А теперь это новый объект string"; // 6
Console.WriteLine("s3 = " + s3);
Console.WriteLine("s4 = " + s4);
Console.WriteLine(Object.ReferenceEquals(s3, s4) ?
      "ref s3 == ref s4" : "ref s3 != ref s4"); // 7

string s5 = "c:\\path\\file.ext"; // 8
string s6 = @"c:\path\file.ext"; // 9

Console.WriteLine("s5 = " + s5);
Console.WriteLine("s6 = " + s6);

string s7 = new string('!', 10); // 10

Console.WriteLine("s7 = " + s7);

char[] cstr = "строка C++".ToCharArray(); // 11
string s8 = new string(cstr); // 12
string s9 = new string(cstr, 0, 8); // 13

Console.WriteLine("s8 = " + s8);
Console.WriteLine("s9 = " + s9);
```

Результат работы кода:

```
s1 =
s2 = Hello!
s3 = Hello!
s4 = Hello!
ref s3 == ref s4
s3 = Hello!
s4 = А теперь это новый объект string
ref s3 != ref s4
s5 = c:\path\file.ext
s6 = c:\path\file.ext
s7 = !!!!!!!!!!
s8 = строка C++
s9 = строка C
```

Разберем подробнее строки, помеченные комментариями:

1) Строка s1 создается пустой.

2) Строка s2 создается вызовом конструктора с параметром типа **char**[] (массив **char**). Т.к. строка «"Hello!"» является экземпляром типа **string**, а среди конструкторов класса String нет конструктора с аргументом такого типа, преобразуем **string** в массив **char** (такой конструктор есть).

3) Но гораздо проще использовать оператор «=» – строка s3 инициализирована строкой «Hello!».

4) Строки – ссылочный тип, поэтому s4 будет ссылкой на экземпляр s3. Вернее, обе эти ссылки указывают на один и тот же объект. Это позволяет экономить память при работе со строками. Следующие операторы выводят на экран строку s1 (пустую) и остальные, содержащие «Hello!».

5) Убеждаемся, что ссылки s3 и s4 указывают на один и тот же объект.

6) Как только мы модифицируем содержимое строки s4, в памяти создается новый экземпляр класса String (содержащий строку «А теперь это новый объект string»), и ссылка s4 теперь указывает на него. Выводим на консоль строки s3 и s4, видим, что теперь они различаются.

7) Убеждаемся, что ссылки s3 и s4 теперь указывают на два разных объекта.

8) Помещаем в строку путь к файлу. Учитывая, что обратный слеш используется для обозначения управляющих символов, ставим слеши парами.

9) Если перед строковым литералом поставить символ «@», то все управляющие символы будут рассматриваться как обычные символы строки. Теперь достаточно одинарных обратных слешей. Далее выводим на консоль строки s5 и s6, убеждаемся, что их содержимое идентично.

10) Еще один вариант конструктора класса String, создает строку из указанного количества элементов типа **char**. В данном случае получаем строку s7, состоящую из десяти восклицательных знаков.

11) Используя метод String.ToCharArray, получаем символы строки «"строка C++"» в виде массива **char**.

12) Вызываем тот же конструктор, что и в строке 2, получаем экземпляр s8.

13) Если в конструкторе указать еще два целых числа, то строка s9 будет инициализирована не всем массивом cstr. Первое число – стартовый элемент массива, второе – количество элементов. В результате строка s9 будет содержать «строка C». Последующий вывод на консоль позволяет убедиться в этом.

Как видно из таблицы Б.1, строки поддерживают операцию конкатенации (+). Мы этим часто пользовались и ранее, когда выводили данные на консоль. Если при использовании оператора «+» один из аргументов имеет тип **string**, а второй – нет, то второй аргумент преобразуется к типу **string** неявным вызовом метода ToString. Однако, надо помнить, что сложение в языке C# выполняется слева направо (см. § 3.3), поэтому в данном примере:

```
Console.WriteLine(s8 + " " + 1 + 2 + 3);
Console.WriteLine(1 + 2 + 3 + " " + s8);
```

на консоли увидим следующее:

```
строка C++ 123
6 строка C++
```

В первом случае к строке последовательно добавлялись числа, будучи также предварительно преобразованными к строкам. Во втором случае сначала складывались числа, и только потом результат их сложения (6) был сцеплен со строкой. Если такой результат нежелателен, необходимо использовать явное преобразование к строке:

```
Console.WriteLine(1.ToString() + 2 + 3 + " " + s8);
```

В этом случае будет использоваться конкатенация строк:

```
123 строка C++
```

Если внимательно изучить методы и операторы класса String, можно заметить, что ни один из них строку не модифицирует – при проведении над строкой любой операции, изменяющей ее содержимое, результат возвращается в виде новой строки, а исходная строка остается без изменений. И попытка модификации строки через индексатор вызовет ошибку компиляции:

```
string s10 = "Hello!";

s10[5] = '?'; // Ошибка
```

Подробнее об индексаторах мы будем говорить в § 4.5. Что же необходимо сделать, чтобы изменить содержимое строки? Если у нас есть очень большая строка, а в ней требуется модифицировать всего один символ (пусть его позиция хранится в переменной spos), мы вынуждены будем сделать следующее:

```
string s11 = "Очень длинная строка";
int slen = s11.Length;
int spos = 9;

s11 = s11.Substring(0, spos) + 'H' + s11.Substring(spos + 1, slen - spos
```

```
- 1);
      Console.WriteLine(s11);
```

В итоге получим строку «Очень длиНная строка». Но данный код, мало того, что не очень наглядный, еще и очень расточительный в плане ресурсов – в процессе его работы выделяется память для четырех строк:

1. Сначала для строки «Очень длинная строка»;

2. Затем, после вызова Substring(0, spos), создается новая строка «Очень дли»;

3. Далее к ней прибавляется символ «Н», получается новая строка «Очень длиН»;

4. Затем добавляем s11.Substring(spos + 1, slen – spos – 1), и получаем окончательный результат «Очень длиНная строка».

Рано или поздно сборщик мусора удалит временные объекты-строки, на которые не осталось ссылок, а это потребует еще некоторого количества дополнительных ресурсов.

Строки с возможностью модификации

Для решения подобных проблем в языке C# есть еще один класс для работы со строками, допускающими модификацию содержимого – это класс System.Text.StringBuilder. Основные его члены приведены в табл. Б.2 (приложение Б).

Теперь изменить один символ в строке очень легко:

```
      StringBuilder s12 = new StringBuilder("Очень длинная строка");

      s12[9] = 'H';
      Console.WriteLine(s12);
```

Класс StringBuilder имеет некоторые особенности. Во-первых, он описан с модификатором sealed (см. § 4.2), поэтому от него нельзя наследоваться. Во-вторых, раз строка может модифицироваться, следовательно, ее длина непостоянна. Если же длина строки меняется, это ведет к фрагментации памяти, да и на регулярное выделение и освобождение памяти тратятся ресурсы ПК. Поэтому в классе StringBuilder есть такое понятие, как *емкость*. Емкость говорит о том, сколько реально выделено места для хранения строки. Текущая длина строки всегда меньше либо равна ее емкости. Пример:

```
      StringBuilder s13 = new StringBuilder("Hello!");

      Console.WriteLine("емкость: " + s13.Capacity);
```

```
Console.WriteLine("длина: " + s13.Length);
for (int i = 0; i < 10; i++) s13.Append('!');
Console.WriteLine("емкость: " + s13.Capacity);
Console.WriteLine("длина: " + s13.Length);
Console.WriteLine(s13);
s13.Append('!');
Console.WriteLine("емкость: " + s13.Capacity);
Console.WriteLine("длина: " + s13.Length);
```

Результат работы кода:

```
емкость: 16
длина: 6
емкость: 16
длина: 16
Hello!!!!!!!!!!
емкость: 32
длина: 17
```

Видно, что изначально была выделена память для 16 символов, хотя строка состояла из шести. Зато потом, когда мы в цикле прибавляем к строке новые символы, перераспределение памяти не требуется. В итоге строка заполняет всю доступную емкость. Если попытаться добавить к строке еще один символ, ее емкость удвоится.

В целом, рекомендация такая. Если ожидается, что в строку будут помещены 10000 символов, но не сразу, а постепенно (например, при чтении из файла и т.п.), то имеет смысл заранее задать сопоставимую емкость строки. При интенсивной работе со строками это повысит общую производительность приложения.

Подобного принципа придерживаются и некоторые другие классы с динамически меняющимся содержимым.

3.1.3.2. Массивы

В языке C# массивы любого вида являются объектами, производными от базового класса System.Array. Поэтому, хотя синтаксис определения массива напоминает синтаксис C++ или Java, реально мы создаем при этом экземпляр класса .NET. И каждый объявленный массив наследует члены класса System.Array.

Данный класс мы рассмотрим несколько позже. Основные его члены представлены в табл. B.1 (приложение B).

 Пример: Samples\3.1\3_1_3_array.

Одномерные массивы

Для объявления одномерного массива на C# нужно поместить пустые квадратные скобки между именем типа и именами переменных:

```
<тип>"[]" {<имя переменной> [ = <инициализатор>]} [, ...];
```

Этот синтаксис отличен от синтаксиса языка C++, в котором квадратные скобки идут после имени переменной. Поскольку любой массив – это экземпляр класса, многие из правил объявления переменных ссылочного типа применяются и к массивам. Например, при объявлении массива на самом деле мы не создаем его. Мы должны явно создать экземпляр массива при помощи оператора **new**, и только после этого он будет существовать в том смысле, что для его элементов будет выделена память. Например:

```
int[] x1 = new int [3];
```

В языке C++ существует возможность инициализации элементов массива при его объявлении (если, конечно, память под него не выделяется динамически), например:

```
int x[3] = { 1, 2, 3 };
int y[] = { 5, 6, 7, 8 };
```

В последнем случае компилятор сам подсчитывает количество элементов в списке инициализации, поэтому массив «y» будет иметь размер 4 элемента. Аналогичная возможность существует и в языке C#, за некоторыми ограничениями. В языке C++, если при инициализации переменной «x» указать меньше инициализаторов, чем элементов в массиве, оставшиеся элементы будут заполнены нулями. В языке C# количество инициализаторов должно совпадать с размером массива:

```
int[] x1 = new int [3] { 1, 2, 3 };
int[] y1 = new int [] { 5, 6, 7, 8 };
```

Более того, если при инициализации задается список элементов, оператор **new** можно опускать:

```
int[] x2 = { 1, 2, 3 };
int[] y2 = { 5, 6, 7, 8 };
```

Последний вариант синтаксиса можно использовать только при создании массива (т.е. когда объявляется экземпляр массива и/или используется оператор **new**):

```
int[] z1 = new int[] { 1, 2 }; // OK
```

```
int[] z2 = { 1, 2 }; // OK
int[] z3, z4;

z3 = new int[] { 1, 2 }; // OK
z4 = { 1, 2 }; // Ошибка
```

Еще один вариант инициализации массива – это инициализация другим экземпляром:

```
z4 = z3;
z3[0] = 123;
Console.WriteLine("z4[0] = " + z4[0]); // z4[0] = 123
z4 = (int[])z3.Clone();
z3[0] = 321;
Console.WriteLine("z4[0] = " + z4[0]); // z4[0] = 123
```

Надо только помнить, что при присваивании копируется ссылка. Поэтому, если мы в данном примере изменяем элементы массива z3, то тем самым мы изменяем и элементы массива z4. Если мы хотим копировать не ссылки, а содержимое массива, то можно использовать метод Clone. Видно, что после этого ссылки указывают на разные массивы в динамической куче.

Как и в языке C++, если N – количество элементов в массиве, то их индексы лежат в диапазоне от 0 до N – 1, а доступ осуществляется при помощи оператора «[]» (который на самом деле является индексатором, см. § 4.5).

Теперь нам понятно, почему аргументы командной строки передаются в функцию Main при помощи конструкции «**string**[] args». Это одномерный массив строк, и узнать его размер можно при помощи свойства Length. Можно вывести его элементы на консоль:

```
Console.WriteLine("Количество аргументов: " + args.Length);

for (int i = 0; i < args.Length; i++)
{
      Console.WriteLine("Аргумент {0}: \"{1}\"", i, args[i]);
}
```

Чтобы задать аргументы командной строки из среды разработки, необходимо зайти в свойства проекта (окно «Проект → Свойства…») и выбрать вкладку «Отладка» (рис. 3.3).

Вводим аргументы, сохраняем изменения и запускаем программу. Вывод этого фрагмента кода будет следующим:

```
Количество аргументов: 3
Аргумент 0: "это"
Аргумент 1: "/мои"
Аргумент 2: "-аргументы"
```

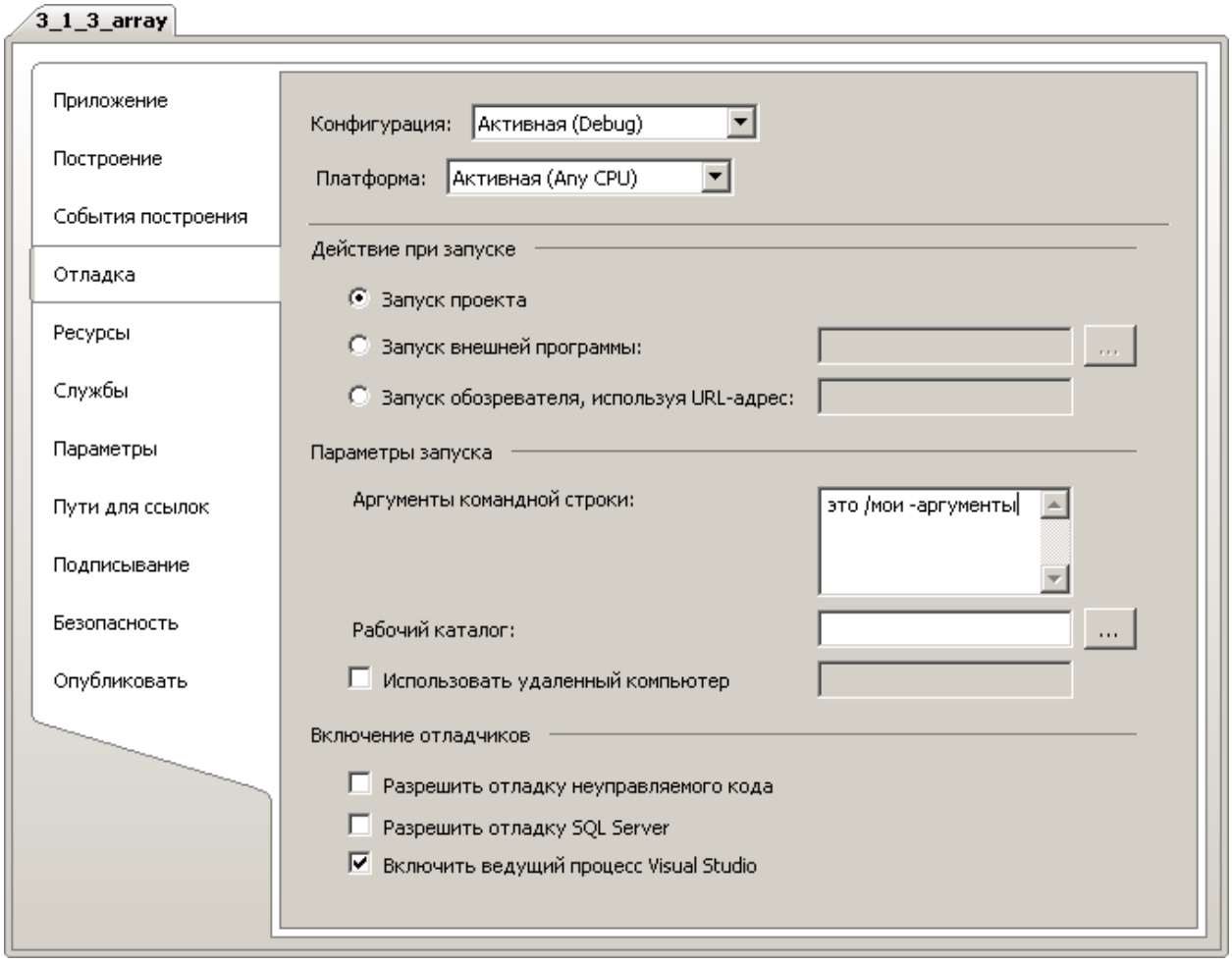


Рис. 3.3 – Аргументы командной строки в Visual Studio 2008 Professional

Прямоугольные массивы

Прямоугольные (или ровные) массивы представляют собой многомерные кубы значений. Элементы таких массивов идентифицируются набором индексов – «координат» в многомерном пространстве. Каждое измерение имеет свою размерность, не зависящую от других. Отметим, что многомерные массивы являются важным отличием от других подобных языков (Java), ибо по сравнению с неровными массивами (рассмотренными ниже), обеспечивают большую производительность.

При декларации размерности измерений указываются через запятую:

```
<тип>"[",[...]"]" {<имя переменной> [ = <инициализатор>]} [, ...];
```

Варианты инициализации те же, что и для одномерных массивов – явный вызов оператора **new** (с инициализацией значениями или без), неявный (при инициализации значениями) либо присвоение ссылки на другой массив.

Доступ к элементам прямоугольного массива также производится с помощью оператора [], в котором индексы указываются через запятую.

Пример:

```
double [,] m = new double [10, 10];
string [,] s = {{"11", "12"}, {"21", "22"}, {"31", "32"}};

Console.WriteLine("Размерность m: " + m.Rank);
Console.WriteLine("m: " + m.GetLength(0) + "x" + m.GetLength(1));
Console.WriteLine("Элементов в s: " + s.Length);
Console.WriteLine("s: " + s.GetLength(0) + "x" + s.GetLength(1));

for (int i = 0; i < s.GetLength(0); i++)
{
      for (int j = 0; j < s.GetLength(1); j++)
      {
            Console.Write(s[i, j] + " ");
      }
      Console.WriteLine();
}
```

Свойство Length возвращает суммарное число элементов массива, поэтому в данном примере это свойство для массива «s» вернет 6. Для определения размера каждого измерения массива в методе используется метод GetLength. Число измерений массива называется рангом, а его значение позволяет получить свойство Rank.

Вывод на консоль:

```
Размерность m: 2
m: 10x10
Элементов в s: 6
s: 3x2
11 12
21 22
31 32
```

Ортогональные массивы

Ортогональные, или неровные (jagged) массивы – это, по сути, массивы массивов. Собственно, формы декларации и доступа вытекают из этого:

```
<тип>"[]""[]"[...] {<имя переменной> [ = <инициализатор>]} [, ...];
```

Т.е. если в прямоугольных массивах количество измерений указывалось количеством запятых, то в данном случае – количеством квадратных скобок. Для доступа к элементам ортогонального массива каждый индекс также стоит в отдельных скобках.

Пример:

```
int[][] ort = new int[3][];
```

```
ort[0] = new int[4];
ort[1] = new int[] { 11, 22, 33, 44, 55 };
ort[2] = new int[] { 1, 2, 3 };
Console.WriteLine("Размерность ort: " + ort.Rank);
Console.WriteLine("Элементов в ort: " + ort.Length);

for (int i = 0; i < ort.Length; i++)
{
      for (int j = 0; j < ort[i].Length; j++)
      {
            Console.Write(ort[i][j] + " ");
      }
      Console.WriteLine();
}
```

Вывод на консоль:

```
Размерность ort: 1
Элементов в ort: 3
0 0 0 0
11 22 33 44 55
1 2 3
```

Как мы видим, массив «ort» считается одномерным. Просто его элементами являются другие массивы. Отсюда и различия в синтаксисе с прямоугольными массивами. В массиве ort[0] элементы не инициализированы, поэтому имеют значения по умолчанию (для типа **int** это 0). Описание массива ort можно было сделать и одной строкой:

```
int[][] ort = { new int[4], new int[] { 11, 22, 33, 44, 55 },
                new int[] { 1, 2, 3 } };
```

Класс System.Array

Класс Array является абстрактным (см. главу 4). Это означает, что запрещено создавать его экземпляры. Можно создавать только экземпляры производных классов, перегружающих его абстрактные методы. Основные члены класса Array представлены в табл. В.1 (приложение В).

Все остальные массивы (одномерные, прямоугольные и ортогональные) являются потомками класса Array (рис. 3.4). Сам класс Array наследуется от System.Object и реализует четыре интерфейса – ICollection, ICloneable, IEnumerable и IList (см. п. 4.9.4).

Очевидно, что вариантов создания массивов очень много (в зависимости от типа элементов, количества размерностей и, в случае с ортогональными массивами – длиной каждого измерения), и невозможно предусмотреть отдельный класс для каждого случая. Как же компилятор решает эту пробле-

му?

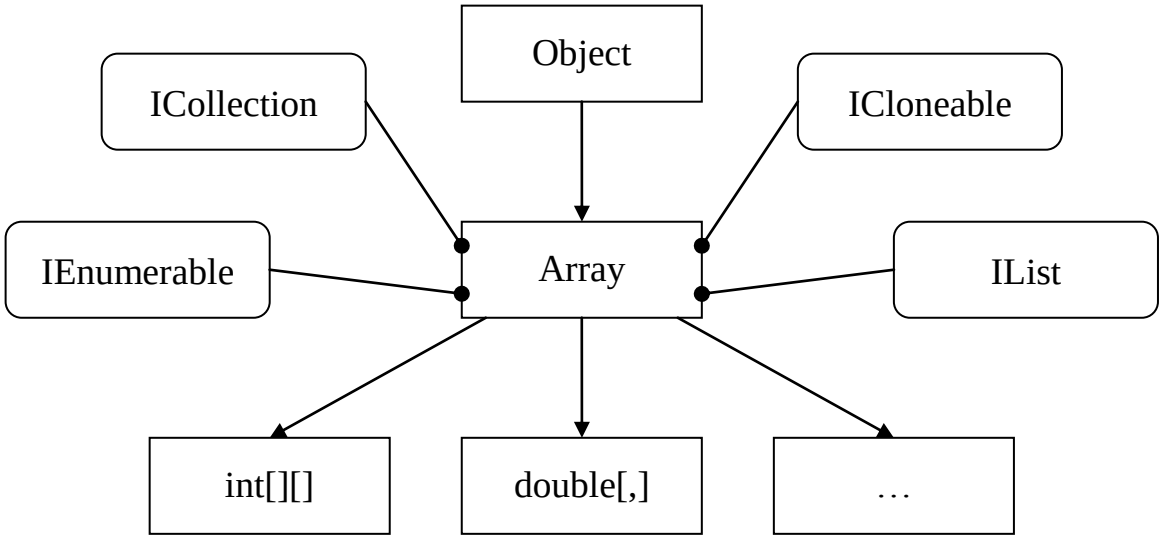


Рис. 3.4 – Иерархия классов массивов в языке C#

На самом деле, любой массив создается как экземпляр класса Array, но компилятор предоставляет им некую обертку, которая позволяет им считаться потомками класса Array:

```
Console.WriteLine("Класс типа double[,]: " + m);
Console.WriteLine("Базовый класс для double[,]: " +
      m.GetType().BaseType);
Console.WriteLine("Класс типа int[][]: " + ort);
Console.WriteLine("Базовый класс для int[][]: " +
            ort.GetType().BaseType);
```

Результат работы данного фрагмента кода:

```
Класс типа double[,]: System.Double[,]
Базовый класс для double[,]: System.Array
Класс типа int[][]: System.Int32[][]
Базовый класс для int[][]: System.Array
```

Т.к. класс Array является абстрактным, невозможно создать его экземпляр вызовом конструктора. Поэтому для создания любого массива используется метод Array.CreateInstance:

```
Array dyn1 = Array.CreateInstance(typeof(double), 4, 5);

Console.WriteLine("dyn1: " + dyn1);

for (int i = 0; i < dyn1.GetLength(0); i++)
{
      for (int j = 0; j < dyn1.GetLength(1); j++)
      {
            Console.Write(dyn1.GetValue(i, j) + " ");
      }
      Console.WriteLine();
}
```

```
Array dyn2 = Array.CreateInstance(typeof(int[]), 2);

Console.WriteLine("dyn2: " + dyn2);
dyn2.SetValue(new int[3], 0);
dyn2.SetValue(new int[4], 1);

for (int i = 0; i < dyn2.Length; i++)
{
      for (int j = 0; j < ((int[])dyn2.GetValue(i)).Length; j++)
      {
            Console.Write(((int[])dyn2.GetValue(i)).GetValue(j) + " ");
      }
      Console.WriteLine();
}
```

Результат работы:

```
dyn1: System.Double[,]
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
dyn2: System.Int32[][]
0 0 0
0 0 0 0
```

В данном примере при помощи метода CreateInstance были созданы массивы типа **double**[,] и **int**[][].

3.1.4. Анонимные типы

Объявляемые в области метода переменные могут иметь неявный тип **var**. Локальная переменная с неявным типом имеет строгую типизацию, как если бы тип был задан явно, только тип определяет компилятор. Следующие два объявления переменной «i» функционально являются эквивалентами:

```
var i = 10; // неявная (implicitly) типизация
int i = 10; // явная (explicitly) типизация
```

При этом ключевое слово **var** является контекстно-зависимым. Оно является ключевым только в контексте описания типа локальной переменной. Во всех остальных участках программы var – обычный идентификатор. В частности, можно объявить локальную переменную с именем var, даже таким образом:

```
var var = 5;
```

Здесь первое вхождение var является ключевым словом, второе – идентификатором.

Переменная анонимного типа обязательно должна быть инициализиро-

вана в момент объявления. Именно по типу инициализирующего значения определяется тип объявляемой переменной:

```
      static int Main(string[] args)
      {
            var v1 = 15;
            var v2 = 15u;
            var v3 = 'X';
            var v4 = "АБВ";
            var v5 = new float[] { 1f, 2f, 3f };
            var v6 = args;
            var v7; // Ошибка

            Console.WriteLine(v1.GetType());
            Console.WriteLine(v2.GetType());
            Console.WriteLine(v3.GetType());
            Console.WriteLine(v4.GetType());
            Console.WriteLine(v5.GetType());
            Console.WriteLine(v6.GetType());
            return 0;
      }
```

Результаты работы:

```
      System.Int32
      System.UInt32
      System.Char
      System.String
      System.Single[]
      System.String[]
```

Наиболее важное применение анонимная типизация имеет при создании анонимных типов объектов в выражениях запросов LINQ (Language-Integrated Query). Это инструмент, позволяющий писать запросы (подобно запросам SQL к базам данных) к коллекциям данных прямо в тексте программы. Справку по LINQ можно получить в справочной системе MSDN. Небольшой пример:

```
      int[] nums = { 1, 2, 15, 13, 4, 6, 7, 1, 9, 10 };
      var odd = from n in nums
                where n % 2 == 1
                select new { Description = "Найдено нечётное число: {0}",
Number = n };

      foreach (var num in odd)
      {
            Console.WriteLine(num.Description, num.Number);
      }
```

Результаты работы:

```
      Найдено нечётное число: 1
      Найдено нечётное число: 15
      Найдено нечётное число: 13
      Найдено нечётное число: 7
```

```
Найдено нечётное число: 1
Найдено нечётное число: 9
```

Пример: Samples\3.1\3_1_4_var.

3.1.5. Упаковка

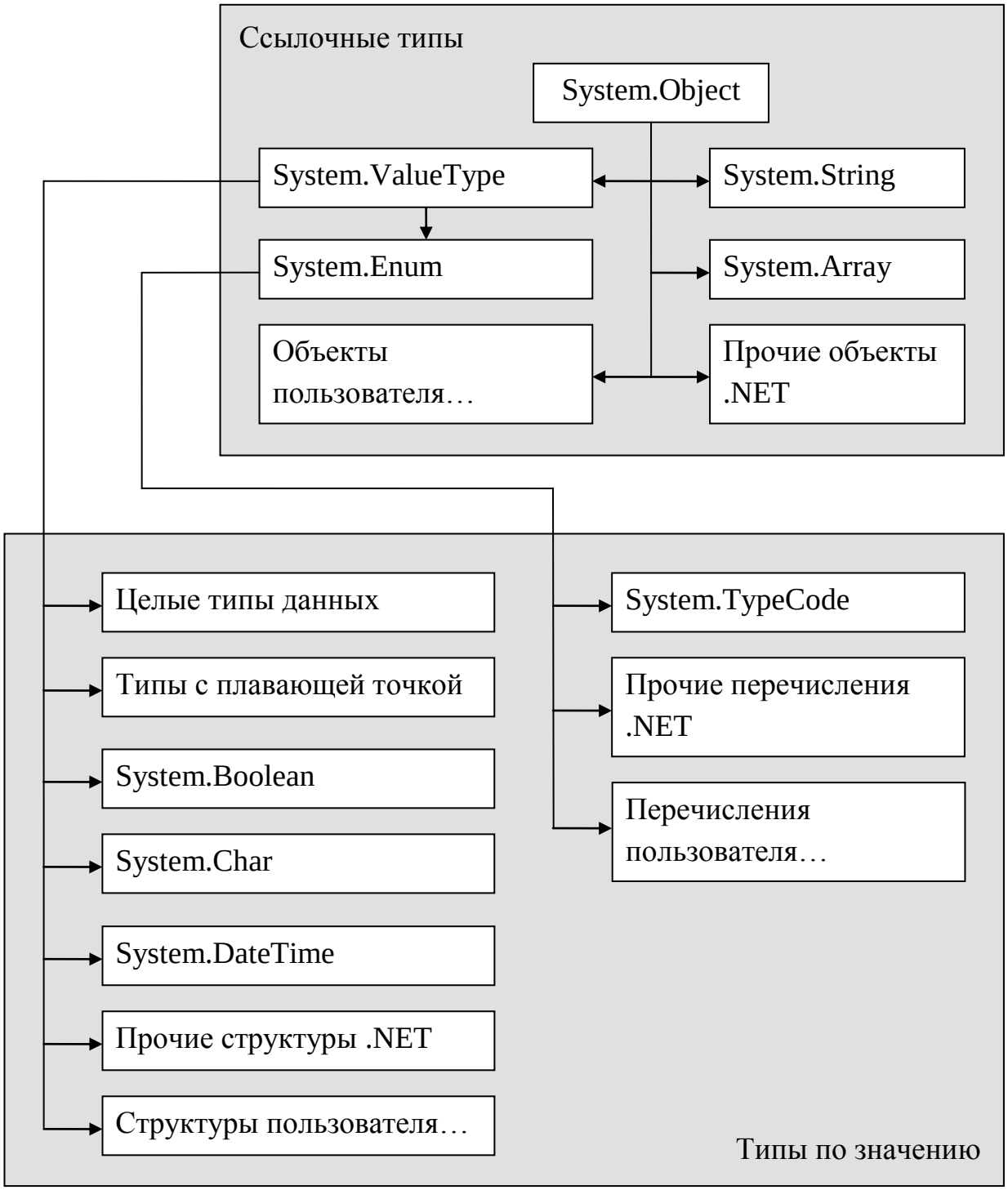


Рис. 3.5 – Структура типов .NET

Как мы могли заметить, типы по значению проявляют дуализм в том смысле, что:

а) это значения, хранящиеся на стеке;

б) это классы, для которых мы можем вызывать методы, а классы – ссылочные типы, хранящие данные в динамической куче.

в) типы по значению, например, **int**, наследуются от класса Object, типа по ссылке (рис. 3.5).

Как такое возможно? Для каждого типа по значению поддерживается соответствующий «упакованный» (boxed) тип, который является классом, реализующим то же поведение и содержащим те же данные. Если требуется передать тип значения по ссылке, он автоматически упаковывается (box) в соответствующий упакованный тип, а по прибытии при необходимости распаковывается (unbox) снова в тип значения. Находясь в упакованном виде, тип может использовать все методы класса System.ValueType. Например, допустима следующая конструкция:

```
string s = 54.ToString();
```

При этом значение 54 упаковывается в класс System.Int32, наследующий от класса System.ValueType метод ToString(), который и вызывается.

Другие примеры:

```
Console.WriteLine(1ul.GetType());
Console.WriteLine("Hello!".Length);
```

В данном случае на консоли отобразится имя класса System.UInt64 и длина строки «Hello!» – 6 символов.

3.1.6. Переменные и идентификаторы

В языке C# определено несколько типов переменных: статические, экземплярные, элементы массивов, параметры по значению, параметры по ссылке, выходные параметры и локальные переменные.

С массивами мы уже познакомились, а здесь разберем лишь локальные переменные. Все остальные типы переменных будут рассмотрены позже.

3.1.6.1. Идентификаторы

Имена в программах служат той же цели, что и имена в мире людей – чтобы обращаться к объектам и различать их, то есть идентифицировать. По-

этому имена также называют идентификаторами. В идентификаторе могут использоваться буквы, цифры и символ подчеркивания. Компилятором языка C# различаются прописные и строчные буквы, поэтому sysop, SySoP и SYSOP – три разных идентификатора. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Длина идентификатора не ограничена. Пробелы внутри имен не допускаются. В идентификаторах C# разрешается использовать, помимо латинских букв, буквы национальных алфавитов. Например:

```
enum Радуга { Красный, Оранжевый, Желтый, Зеленый, Голубой,
              Синий, Фиолетовый }
int Счётчик;
```

Более того, в идентификаторах можно применять даже символы Unicode:

```
int \u005fid;
int _id; // Ошибка
```

В этом примере в первой строке объявлен идентификатор «_id», т.к. U+005F – это код символа подчеркивания. Поэтому для второй строки компилятор выдаст ошибку – дублирование идентификатора.

Имена даются элементам программы, к которым требуется обращаться: переменным, типам, константам, методам, меткам и т.д. Идентификатор создается на этапе объявления переменной (метода, типа и т.п.), после этого его можно использовать в последующих операторах программы.

При выборе идентификатора необходимо иметь в виду, что он не должен совпадать с ключевыми словами языка. Однако, сборки .NET могут включать модули, написанные на различных языках программирования. Что, если мы описали переменную, например, с именем input, которое не является ключевым словом C#, но может являться ключевым словом другого языка? Или переносим фрагмент кода, например, из программы на языке C++, где имеются переменные lock (ключевое слово в C#) и т.д., и не хотим переименовывать идентификаторы? В этих случаях достаточно идентификатор предварить символом «@». Например, правильным будет идентификатор @lock.

3.1.6.2. Локальные переменные

Локальные переменные – это переменные, объявленные в теле метода. Они не инициализируются автоматически, а время их жизни зависит от ком-

пилятора, но оно не больше, чем время исполнения метода. Локальные переменные могут быть определены и в управляющих блоках (for, while, foreach, if). Локальная переменная и поле класса могут иметь одинаковые идентификаторы:

```
class Program
{
      static int i = 123;

      public static int Main(string[] args)
      {
            for (int i = 0; i < 5; i++)
            {
                  Console.WriteLine(i);
            }

            Console.WriteLine(Program.i);
            return 0;
      }
}
```

Здесь переменная «i» объявлена в классе как статическая, т.к. из статических методов класса можно получить доступ только к статическим полям (см. § 4.3). Вывод программы следующий:

```
0
1
2
3
4
123
```

Тип переменной выбирается, исходя из диапазона и требуемой точности представления данных. Например, нет необходимости заводить переменную вещественного типа для хранения величины, которая может принимать только целые значения, хотя бы потому, что целочисленные операции выполняются гораздо быстрее.

Так называемая область действия локальной переменной, то есть область программы, где можно использовать переменную, начинается в точке ее описания и длится до конца блока, внутри которого она описана. Блок – это код, заключенный в фигурные скобки. Основное назначение блока – группировка операторов. Здесь мы говорим только о блоках внутри методов. Про поля, объявленные в классе, будем говорить позже.

Имя переменной должно быть уникальным в области ее действия. Область действия распространяется на вложенные блоки, из этого следует, что и во вложенном блоке нельзя объявить переменную с таким же именем, что и в

охватывающем его, например:

```
class X
{
	int A; // поле A класса X
	int B; // поле B класса X

	void Y() // метод Y класса X
	{
		// локальная переменная C, область действия - метод Y
		int C;
		// локальная переменная A (не конфликтует с полем A)
		int A;
		{
			// локальная переменная D, область действия –
			// этот блок
			int D;
			int A; // Ошибка
		}
		{
			// локальная переменная D, область действия –
			// этот блок
			int D;
		}
	}
}
```

Давайте разберемся в этом примере. В нем описан класс X, содержащий три элемента: поле A, поле B и метод Y. Непосредственно внутри метода Y заданы две локальные переменные – C и A. Внутри метода класса можно описывать переменную с именем, совпадающим с полем класса, потому что существует способ доступа к полю класса с помощью ключевого слова **this** (см. главу 4). Таким образом, локальная переменная A не «закрывает» поле A класса X, а вот попытка описать во вложенном блоке другую локальную переменную с тем же именем окончится неудачей. Две переменные с именем D не конфликтуют между собой, поскольку блоки, в которых они описаны, не вложены один в другой.

Отдельно остановимся на описании переменных внутри управляющих конструкций языка. В некоторых компиляторах языка C++ данный код считается правильным, в других – нет:

```
for(int i = 0; i < 10; i++)
{
	// тело цикла
}

int i;
```

Все зависит от того, на каком уровне вложенности считается объявленной переменная «i» в цикле. Если на том же уровне, что и сам цикл, компиля-

тор выдаст ошибку, если на уровне внутри цикла (т.е. на уровне тела цикла) – то нет. В любом случае, такой код будет компилироваться без ошибки:

```
for(int i = 0; i < 10; i++)
{
      int j;
      // тело цикла
}

int j;
```

В языке C# не будут компилироваться оба варианта. Считается, что второе объявление переменной «j» сделано на один уровень выше, чем первое, и поэтому имена конфликтуют (как и в примере с классом X выше):

```
double[] arr = { 1, 2, 3 };

foreach (double x in arr) Console.WriteLine(x);

int x; // Ошибка

for (int i = 0; i < 5; i++)
{
      int z;
}

int z; // Ошибка
```

Если же второе объявление переменной будет в другом блоке, ошибки не будет:

```
for (int i = 0; i < 5; i++)
{
      Console.WriteLine(i);
}

for (int i = 0; i < 5; i++)
{
      int z;
}

{
      int z;
}
```

В данном примере переменные «i» и «z» объявлены дважды, но не конфликтуют, т.к. находятся в разных ответвлениях блоков кода. Некоторые компиляторы C++ для второго объявления переменной «i» выдали бы ошибку.

Как правило, переменным с большой областью действия даются более длинные и осмысленные имена (описывающие назначение переменной), а для переменных, вся «жизнь» которых – несколько строк исходного текста,

хватит и одной буквы.

3.1.6.3. Константы

Можно запретить изменять значение переменной, задав при ее описании ключевое слово **const**, например:

```
const string s = "123";
const int a = 1;

s = "!!!"; // Ошибка
a = 123; // Ошибка
```

Такие величины называют именованными константами, или просто константами. Они применяются для того, чтобы вместо значений констант можно был использовать в программе их имена. Это делает программу более понятной и облегчает внесение в нее изменений, поскольку изменить значение достаточно только в одном месте программы.

Улучшение читабельности происходит только при осмысленном выборе имен констант. Считается, что в хорошо написанной программе вообще не должно встречаться иных чисел, кроме 0 и 1, все остальные числа должны задаваться именованными константам с именами, отражающими их назначение. Именованные константы должны обязательно инициализироваться при описании. При инициализации можно использовать не только константу, но и выражение, главное, чтобы оно было вычисляемым на этапе компиляции, например

```
const string s = "123" + "!!!"; // "123!!!"
const int a = 1;
const int b = a + 1; // 2
const double d; // Ошибка
```

§ 3.2. Форматирование. Консольный ввод и вывод

Хотя время консольных приложений уходит, многие программы, не требующие взаимодействия с пользователем, остаются консольными. Кроме того, очень часто требуется значения различных типов данных преобразовать в строку со специфическим форматом, или наоборот, извлечь из форматированной строки некоторые значения (осуществить разбор строки). При этом применяются те же инструменты форматирования и разбора, что и при консольном вводе-выводе. Также на таких программах удобно делать первые шаги при обучении новому языку.

Для работы с консолью в .NET используется класс Console. Удобство использование этого класса кроется в двух аспектах: все его методы являются статическими, так что не нужно создавать его экземпляр для использования. Во-вторых, он объединяет в себе ввод, вывод и анализ ошибок. По умолчанию ввод/вывод производится на стандартную консоль (если ее нет, например, в оконных приложениях, вывод просто игнорируется), но устройства ввода и вывода можно изменить.

Для работы с консолью обычно используется три метода – ReadLine, Write и WriteLine. Первый используются для ввода, последние два – для вывода. Но сначала нужно разобраться, как форматировать строки и как осуществлять разбор форматированных строк.

3.2.1. Методы Format и ToString

Для форматирования данных мы можем использовать несколько методов:

1) Перегруженный метод Object.ToString();

2) Собственный метод ToString(**string** format), определенный для каждого типа данных по значению (кроме типов **bool**, **char**, **string**);

3) Метод ToString (IFormatProvider provider), наследуемый типами данных по значению от интерфейса IConvertible;

4) Метод ToString (**string** format, IFormatProvider provider), наследуемый типами данных по значению от интерфейса IFormattable (кроме типов **bool**, **char**, **string**);

5) Ряд статических методов String.Format с различным набором аргументов;

6) Статический метод Enum.Format (Type enumType, **object** value, **string** format) для элементов перечисляемого типа.

Первые четыре метода позволяют форматировать любые типы данных, кроме структур (т.к. они не имеют своего класса, а наследуются напрямую от System.ValueType, и их поля необходимо форматировать по отдельности), а также типов **bool**, **char** и **string** (к ним применимы только первый и третий вариант метода). Хотя для структур пользователя можно перегрузить метод Object.ToString (о том, как это сделать см. § 4.7) и написать другие варианты метода ToString с желаемым набором параметров. В частности, для структуры DateTime применимы все описанные варианты данного метода. Пятый метод позволяет форматировать любые типы данных, кроме структур (также за исключением DateTime). Шестой метод применим только для перечислений.

 Пример: Samples\3.2\3_2_1_format.

3.2.1.1. Интерфейс IFormatProvider

Интерфейс IFormatProvider предоставляет механизм извлечения объекта для управления форматированием. Те методы форматирования, которые имеют аргумент типа IFormatProvider, используют его для указания дополнительных опций преобразования значения в строку (такие, как региональные стандарты и т.п.). В структуре классов .NET описаны три класса, реализующие данный интерфейс:

1) System.Globalization.NumberFormatInfo – определяет способ форматирования и отображения числовых значений, зависящий от языка и региональных параметров;

2) System.Globalization.DateTimeFormatInfo – определяет способ форматирования и отображения значений DateTime, зависящих от языка и региональных параметров;

3) System.Globalization.CultureInfo – служит для предоставления сведений об определенном языке и региональных параметрах. Наиболее универсальный класс, т.к. содержит поле DateTimeFormat, являющееся экземпляром класса DateTimeFormatInfo, и поле NumberFormat, являющееся экземпляром класса NumberFormatInfo.

Основные члены этих классов приведены в приложении Г.

Например, в русском языке целая часть числа отделяется от дробной

запятой, а в английском – точкой. Какой символ будет использован по умолчанию? Это определяет свойство CultureInfo.InstalledUICulture – региональные настройки операционной системы:

```
double value = 12345.6789;

Console.WriteLine(value);
Console.WriteLine(value.ToString(CultureInfo.InstalledUICulture));
```

Действительно, в обоих случаях на экран выводится строка «12345,6789», как это принято в русском языке (если, конечно, на Вашем компьютере установлена русская версия ОС Windows).

Хотя операционная система позволяет менять формат отображения чисел, времени и дат. Для этого на панели управления необходимо выбрать элемент «Язык и региональные стандарты» (рис. 3.6).

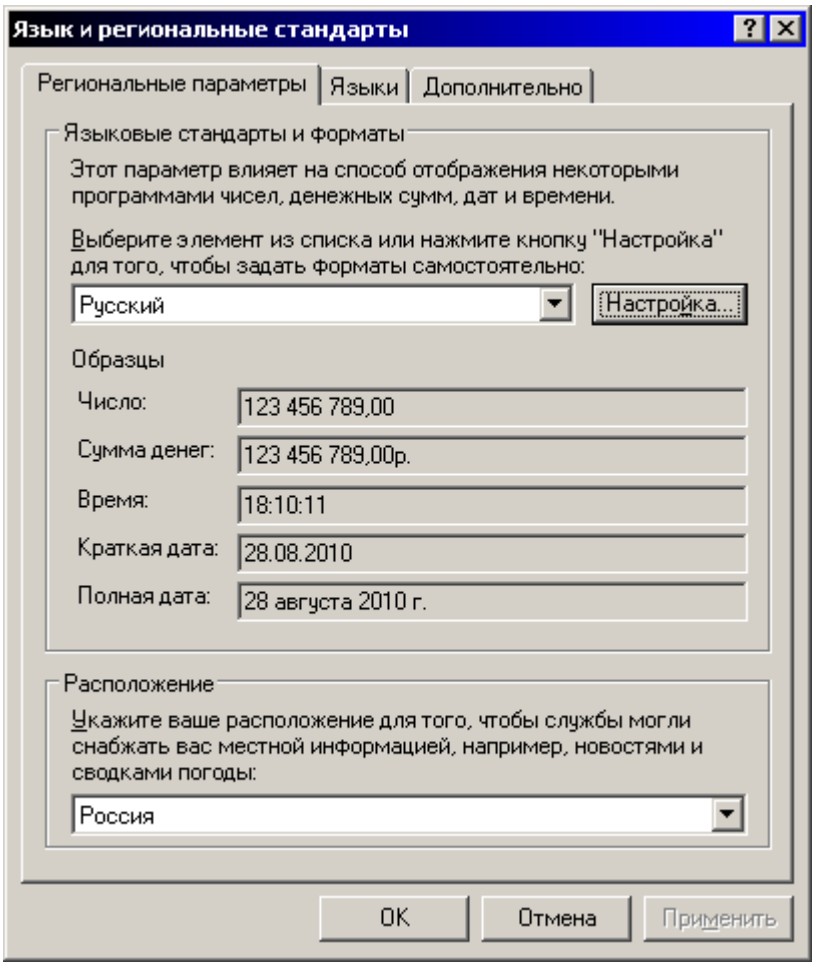


Рис. 3.6 – Региональные параметры ОС Windows XP

В появившемся диалоге для каждого поддерживаемого языка можно посмотреть образцы формата чисел, валюты, времени, а также полной и ко-

роткой даты. Для изменения этих форматов нажмите кнопку «Настройка», появится диалог настройки региональных параметров (рис. 3.7). Эти настройки влияют на все приложения, установленные в системе, поэтому не следует их менять без особой необходимости. Если стоит задача изменить, например, запятую на точку при отображении чисел, можно использовать класс NumberFormatInfo.

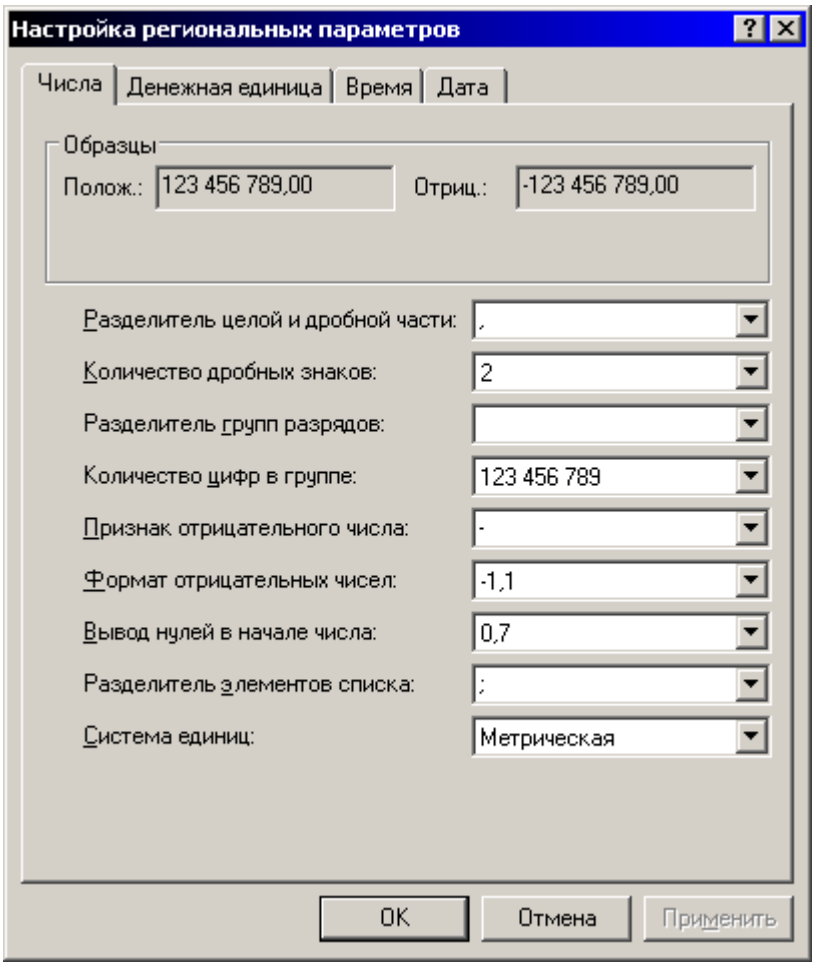


Рис. 3.7 – Настройка региональных параметров ОС Windows XP

Пример:

```
NumberFormatInfo nfi = new NumberFormatInfo();

nfi.NumberDecimalSeparator = ".";
Console.WriteLine(value.ToString(nfi));
nfi.NumberDecimalSeparator = ",";
Console.WriteLine(value.ToString(nfi));
```

В первом случае, независимо от настроек ОС, для разделения целой и дробной части числа будет использована точка (12345.6789), а во втором – запятая (12345,6789).

Мы уже говорили о том, что класс CultureInfo содержит, в том числе, экземпляры классов NumberFormatInfo и DateTimeFormatInfo. Поэтому изменить десятичный разделитель при выводе можно и с его помощью:

```
      CultureInfo ci = (CultureInfo)CultureInfo.InstalledUICulture.Clone();

      ci.NumberFormat.NumberDecimalSeparator = ".";
      Console.WriteLine(value.ToString(ci));
```

Число будет выведено с точкой.

А можно создать свой экземпляр CultureInfo, указав требуемый региональный стандарт. Существует несколько способов сделать это:

1) Клонировать имеющийся вариант региональных стандартов (как это было сделано выше);

2) Создать новый вариант, используя метод CreateSpecificCulture:

```
      Console.WriteLine(value.ToString(CultureInfo.CreateSpecificCulture("en-
GB")));
      Console.WriteLine(value.ToString(CultureInfo.CreateSpecificCulture("ru-
RU")));
```

В первом случае число выведется в формате, специфическом для английского языка (English – Great Britain) (с точкой), во втором – для русского языка (с запятой).

3) Создать новый вариант, используя конструктор класса CultureInfo:

```
      Console.WriteLine(value.ToString(new CultureInfo("en-GB")));
      Console.WriteLine(value.ToString(new CultureInfo("ru-RU")));
```

Результат будет аналогичен предыдущему. Если ожидается многократное использование региональных параметров, то ссылку на созданный экземпляр CultureInfo имеет смысл сохранить:

```
      CultureInfo us = CultureInfo.CreateSpecificCulture("en-US");
      CultureInfo ru = new CultureInfo("ru-RU");

      Console.WriteLine(value.ToString(us));
      Console.WriteLine(value.ToString(ru));
```

В американском диалекте английского языка (English – United States) также в качестве десятичного разделителя используется точка.

3.2.1.2. Метод ToString

Далее мы будем рассматривать только один вариант метода ToString из четырех описанных выше – <тип>.ToString (**string** format, IFormatProvider provider). Почему? Рассмотрим остальные варианты этого метода:

1) Метод Object.ToString() соответствует вызову метода <тип>.ToString с форматом по умолчанию (обычно это формат «G», о чем мы будем говорить далее) и региональными параметрами CultureInfo.InstalledUICulture;

2) Метод <тип>.ToString(**string** format) соответствует вызову метода <тип>.ToString с указанным форматом format, а также региональными параметрами CultureInfo.InstalledUICulture;

3) Метод <тип>.ToString (IFormatProvider provider) соответствует вызову метода <тип>.ToString с форматом по умолчанию, а также указанным интерфейсом управления форматированием provider.

Таким образом, метод <тип>.ToString (**string** format, IFormatProvider provider) наиболее гибкий, а остальные методы ToString – его упрощенные варианты. Исключение составляют классы Boolean, Char и String, для которых этот метод не определен (как и метод <тип>.ToString(**string** format)).

Объявим в нашем проекте следующие переменные, форматированием которых мы будем заниматься далее:

```
byte v1 = 178;
sbyte v2 = -111;
ushort v3 = 50000;
short v4 = -15555;
uint v5 = 4000000000;
int v6 = -2123123123;
ulong v7 = 123456789123456789;
long v8 = -123456789123456789;
float v9 = 123456789.123456789f;
double v10 = 123456789.123456789;
decimal v11 = 123456789.123456789m;
bool v12 = true;
char v13 = 'A';
string v14 = "ABC";
DayOfWeek v15 = DayOfWeek.Friday;
DateTime v16 = new DateTime(2010, 8, 20, 18, 30, 0, 55);
```

3.2.1.3. Предопределенные форматы

Денежный формат

Обозначается буквой «C» или «c» (currency), за которой может следовать целое число – точность (количество знаков после запятой, по умолчанию – 2). На формат вывода оказывают влияние региональные параметры и следующие поля класса NumberFormatInfo (см. табл. Г.1 в приложении Г):

- **int** CurrencyNegativePattern – шаблон для отрицательных чисел (0-15);
- **int** CurrencyPositivePattern – шаблон для положительных чисел (0-3);

• **string** CurrencySymbol – обозначение денежной единицы;
• **int**[] CurrencyGroupSizes – число цифр в группах;
• **string** CurrencyGroupSeparator – разделитель групп;
• **int** CurrencyDecimalDigits – число десятичных разрядов (0-99);
• **string** CurrencyDecimalSeparator – десятичный разделитель;
• **string** NegativeSign – обозначение отрицательного знака.

В табл. 3.12 расшифрованы шаблоны для CurrencyNegativePattern и CurrencyPositivePattern, а также для рассмотренных далее шаблонов NumberNegativePattern, PercentNegativePattern и PercentPositivePattern.

Табл. 3.12 – Шаблоны форматирования чисел

№ шаблона	Для положительной валюты	Для отрицательной валюты	Для отрицательных чисел	Для положительных процентов	Для отрицательных процентов
0	$n	($n)	(n)	n %	–n %
1	n$	–$n	–n	n%	–n%
2	$ n	$–n	– n	%n	–%n
3	n $	$n–	n–	% n	%–n
4		(n$)	n –		%n–
5		–n$			n–%
6		n–$			n%–
7		n$–			–% n
8		–n $			n %–
9		–$ n			% n–
10		n $–			% –n
11		$ n–			n– %
12		$ –n			
13		n– $			
14		($ n)			
15		(n $)			

Здесь «n» – число, «$» – обозначение валюты, «–» – обозначение знака «минус». Пример:

```
Console.WriteLine(v9.ToString("C", CultureInfo.InvariantCulture));
```

```
Console.WriteLine(us.NumberFormat.CurrencySymbol);
Console.WriteLine(v9.ToString("C3", us));
us.NumberFormat.CurrencyGroupSizes = new int[] { 6 };
Console.WriteLine(v9.ToString("C3", us));
Console.WriteLine(ru.NumberFormat.CurrencySymbol);
Console.WriteLine(v4.ToString("C", ru));
ru.NumberFormat.CurrencySymbol = " рублей";
Console.WriteLine(v4.ToString("C", ru));
```

Вывод на консоль:

```
¤123,456,800.00
$
$123,456,800.000
$123,456800.000
р.
-15 555,00р.
-15 555,00 рублей
```

Видим, что в инвариантных региональных настройках применяется нейтральный знак валюты, в настройках США – доллар (причем знак валюты ставится перед суммой), в настройках России – рубль (после суммы).

Данный формат применим только для целых типов и типов с плавающей точкой (т.е. не применим к типам Enum и DateTime). При попытке использовать его для форматирования неподходящего типа компилятор генерирует исключительную ситуацию System.FormatException (это относится и ко всем остальным форматам).

Десятичный формат

Обозначается буквой «D» или «d» (decimal), за которой может следовать целое число – количество позиций, отводимых для строкового представления числа. Неиспользованные позиции заполняются нулями. На формат вывода оказывает влияние поле NumberFormatInfo.NegativeSign.

Пример:

```
Console.WriteLine(v1.ToString("D", us));
Console.WriteLine(v1.ToString("D6", us));
Console.WriteLine(v2.ToString("D", ru));
ru.NumberFormat.NegativeSign = "минус ";
Console.WriteLine(v2.ToString("D6", ru));
```

Вывод на консоль:

```
178
000178
-111
минус 000111
```

Только для целых чисел. Также есть свой формат «D/d» для перечисле-

ний и DateTime, об этом – ниже.

Научный (экспоненциальный) формат

Обозначается буквой «E» или «e» (exponential), за которой может следовать целое число – точность (количество знаков после запятой, по умолчанию – 6). Регистр формата влияет на регистр обозначения экспоненты. На формат вывода (а также на другие форматы для чисел с плавающей точкой) оказывают влияние следующие поля класса NumberFormatInfo:

• **string** PositiveInfinitySymbol – обозначение «+∞»;

• **string** NegativeInfinitySymbol – обозначение «–∞»;

• **string** NaNSymbol – обозначение Not a Number (NaN);

• **string** NumberDecimalSeparator – десятичный разделитель;

• **string** PositiveSign – обозначение положительной экспоненты;

• **string** NegativeSign – обозначение отрицательного знака мантиссы и экспоненты.

Пример:

```
Console.WriteLine(v3.ToString("E", us));
Console.WriteLine(v10.ToString("E", new CultureInfo("fr-FR")));
Console.WriteLine(v10.ToString("E10", ru));
Console.WriteLine(v3.ToString("e4", ru));
Console.WriteLine(nfi.PositiveInfinitySymbol);
Console.WriteLine(nfi.NegativeInfinitySymbol);
Console.WriteLine(nfi.NaNSymbol);
nfi.NaNSymbol = "Неопределенность!";
float x = 0.0f / 0.0f;
Console.WriteLine(x.ToString("E", nfi));
```

Вывод на консоль:

```
5.000000E+004
1,234568E+008
1,2345678912E+008
5,0000e+004
Infinity
-Infinity
NaN
Неопределенность!
```

Видим, что во Франции, как и в России, в качестве десятичного разделителя используется запятая.

Данный формат применим только для целых типов и типов с плавающей точкой.

Формат с фиксированной точкой

Обозначается буквой «F» или «f» (fixed), за которой может следовать целое число – точность (количество знаков после запятой, по умолчанию – значение NumberFormatInfo.NumberDecimalDigits). Также на формат вывода оказывают влияние следующие поля класса NumberFormatInfo:

• **string** NegativeSign – обозначение отрицательного знака;

• **string** NumberDecimalSeparator – десятичный разделитель.

Пример:

```
Console.WriteLine(ru.NumberFormat.NumberDecimalDigits);
Console.WriteLine(v5.ToString("F", ru));
Console.WriteLine(v11.ToString("F3", ru));
Console.WriteLine(v11.ToString("F0", us));
```

Вывод на консоль:

```
2
4000000000,00
123456789,123
123456789
```

Данный формат применим для целых типов и типов с плавающей точкой. Также есть свой формат «F/f» для перечислений и DateTime, об этом – ниже.

Общий формат

Обозначается буквой «G» или «g» (general), за которой может следовать целое число – количество значащих цифр мантиссы для строкового представления числа (т.е. не учитывая экспоненту). При использовании данного формата происходит попытка найти самую короткую запись числа – это может быть целая, фиксированная или экспоненциальная запись числа. Если число не удается вместить в указанное количество позиций, то оно занимает дополнительные позиции (но также по минимуму). Если количество позиций не указано, или указан ноль, используется:

• для Byte или SByte – 3 позиции;

• для Int16 или UInt16 – 5 позиций;

• для Int32 или UInt32 – 10 позиций;

• для Int64 или UInt64 – 19 позиций;

• для Single – 7 позиций;

• для Double – 15 позиций;

• для Decimal – 29 позиций.

Регистр формата влияет на регистр обозначения экспоненты. На формат вывода оказывают влияние следующие поля класса NumberFormatInfo:

• **string** NegativeSign – обозначение отрицательного знака числа;

• **string** NumberDecimalSeparator – десятичный разделитель;

• **int** NumberDecimalDigits – количество цифр дробной части;

• **string** PositiveSign – обозначение положительного знака числа и экспоненты.

Пример:

```
Console.WriteLine(v6.ToString("G", ru));
Console.WriteLine(v6.ToString("G4", ru));
Console.WriteLine(v6.ToString("G20", ru));
Console.WriteLine(v10.ToString("G0", new CultureInfo("fr-FR")));
Console.WriteLine(v10.ToString("g2", new CultureInfo("fr-FR")));
ru.NumberFormat.NegativeSign = "минус ";
Console.WriteLine(v8.ToString("G", ru));
```

Вывод на консоль:

```
-2123123123
-2,123E+09
-2123123123
123456789,123457
1,2e+08
минус 123456789123456789
```

Данный формат применим для целых типов и типов с плавающей точкой. Также есть свой формат «G/g» для перечислений и DateTime, об этом – ниже.

Числовой формат

Обозначается буквой «N» или «n» (numerical), за которой может следовать целое число – точность (количество знаков после запятой, по умолчанию – 2). На формат вывода оказывают влияние следующие поля класса NumberFormatInfo:

• **string** NegativeSign – обозначение отрицательного знака;

• **int** NumberNegativePattern – шаблон для отрицательных чисел (0..4, см. табл. 3.12);

• **int**[] NumberGroupSizes – число цифр в группах;

• **string** NumberGroupSeparator – разделитель групп;

• **int** NumberDecimalDigits – число десятичных разрядов;

• **string** NumberDecimalSeparator – десятичный разделитель.

Пример:

```
Console.WriteLine(v4.ToString("N", us));
ru.NumberFormat.NegativeSign = "° ниже ноля";
ru.NumberFormat.NumberNegativePattern = 3;
Console.WriteLine(v4.ToString("N0", ru));
```

Вывод на консоль:

```
-15,555.00
15 555° ниже ноля
```

Здесь мы использовали шаблон для отрицательных чисел №3 («n–», см. табл. 3.12), а знак «минус» заменили строкой «° ниже ноля».

Данный формат применим только для целых типов и типов с плавающей точкой.

Процентный формат

Обозначается буквой «P» или «p» (percent), за которой может следовать целое число – точность (количество знаков после запятой, по умолчанию – NumberFormatInfo.PercentDecimalDigits). На формат вывода, кроме уже рассмотренных, оказывают влияние следующие поля класса NumberFormatInfo:

• **string** NegativeSign – обозначение отрицательного знака;

• **int** PercentDecimalDigits – число десятичных разрядов;

• **string** PercentDecimalSeparator – десятичный разделитель;

• **string** PercentGroupSeparator – разделитель групп разрядов;

• **int**[] PercentGroupSizes – число цифр в группах;

• **string** PercentSymbol – знак процентов;

• **int** PercentPositivePattern – шаблон для положительных значений (0-3, см. табл. 3.12);

• **int** PercentNegativePattern – шаблон для отрицательных значений (0-11, см. табл. 3.12).

Пример:

```
v9 = 0.2478f;
Console.WriteLine(v9.ToString("P", ru));
Console.WriteLine(v9.ToString("P1", us));
ru.NumberFormat.PercentDecimalSeparator = " целых ";
ru.NumberFormat.PercentSymbol = " сотых процента";
Console.WriteLine(v9.ToString("P", ru));
```

Вывод на консоль:

```
24,78%
24.8 %
24 целых 78 сотых процента
```

Данный формат применим только для целых типов и типов с плавающей точкой.

Формат приемо-передачи (обратного преобразования)

Обозначается буквой «R» или «r» (receiving-transmitting или reconversion), спецификатор точности игнорируется. Этот формат гарантирует, что строка, полученная преобразованием из числового значения, при обратном преобразовании даст то же самое числовое значение. Такое часто требуется при передаче и последующем приеме числовой информации в виде текста.

Как мы уже знаем, число типа **float** способно хранить 7 значащих цифр, а число типа **double** – 15-16. Для избегания потери значащих цифр, при использовании данного формата точность для типа **float** увеличивается до 9 значащих цифр, а для типа **double** – до 17.

На формат вывода, кроме уже рассмотренных, оказывают влияние следующие поля класса NumberFormatInfo:

• **string** NegativeSign – обозначение отрицательного знака;

• **string** NumberDecimalSeparator – десятичный разделитель;

• **string** PositiveSign – обозначение положительного знака.

Пример:

```
Console.WriteLine(v9.ToString("R", ru));
Console.WriteLine(v9.ToString("G", ru));
Console.WriteLine(v10.ToString("R", ru));
Console.WriteLine(v10.ToString("G", ru));
```

Вывод на консоль:

```
0,2478
0,2478
123456789,12345679
123456789,123457
```

Видно, что данный формат действительно наиболее точный.

Данный формат применим только для типов **float** и **double**. Также есть свой формат «R/r» для DateTime, об этом – ниже.

Шестнадцатеричный формат

Обозначается буквой «X» или «x» (hexadecimal), за которой может следовать целое число – количество позиций, отводимых для строкового представления числа. Неиспользованные позиции заполняются нулями. Регистр формата влияет на регистр шестнадцатеричных цифр (A-F или a-f). На формат вывода никакие дополнительные настройки не влияют.

Пример:

```
Console.WriteLine(v7.ToString("X", ru));
Console.WriteLine(v7.ToString("x", ru));
Console.WriteLine(v8.ToString("X20", us));
Console.WriteLine(v8.ToString("x20", us));
```

Вывод на консоль:

```
1B69B4BACD05F15
1b69b4bacd05f15
0000FE4964B4532FA0EB
0000fe4964b4532fa0eb
```

Только для целых чисел.

Форматирование перечисления

Возможные форматы для перечислимого типа приведены в табл. 3.13.

Табл. 3.13 – Форматы перечисления

Формат	Описание
G или g	Если значение равно именованной перечислимой константе, то возвращается имя этой константы, в противном случае возвращается десятичный эквивалент значения
X или x	Представляет значение в шестнадцатеричной системе
D или d	Представляет значение в десятичной форме
F или f	Действует так же, как «G/g», за исключением того, что наличие FlagsAttribute в объявлении Enum не требуется

Указание точности не предусмотрено (при попытке генерируется исключительная ситуация System.FormatException).

Пример:

```
Console.WriteLine(v15.ToString("G"));
DayOfWeek vv = (DayOfWeek)12;
Console.WriteLine(vv.ToString("G"));
Console.WriteLine(v15.ToString("D"));
Console.WriteLine(v15.ToString("X"));
```

```
      FileAttributes va = FileAttributes.Hidden | FileAttributes.Archive;
      Console.WriteLine(va.ToString("G"));
      BitFlags vb = BitFlags.bit1 | BitFlags.bit5;
      Console.WriteLine(vb.ToString("G"));
      Console.WriteLine(vb.ToString("F"));
```

Вывод на консоль:

```
      Friday
      12
      5
      00000005
      Hidden, Archive
      17
      bit1, bit5
```

Здесь использовалось перечисление BitFlags, описанное нами в п. 3.1.2.5 и перечисление атрибутов файла FileAttributes из пространства имен System.IO.

Как уже было отмечено, формат «G» выводит имя именованной константы, а если такой константы не существует, используется формат «D». Перечисление FileAttributes объявлено с атрибутом FlagsAttribute, поэтому для него использование форматов «G» и «F» даст одинаковый результат. В описании перечисления BitFlags данный атрибут указан не был, поэтому формат «G» не может найти константу со значением 17 в перечислении, а формат «F» принудительно рассматривает эту константу как битовую комбинацию.

Замечание. Здесь была использована версия метода ToString без указания интерфейса управления форматированием, т.к. версию с указанием данного интерфейса компилятор Visual Studio 2008 считает устаревшей, и рекомендует использовать именно эту версию. Хотя интерфейс можно и указать, никакого влияния на форматирование он не окажет.

Форматирование даты и времени

Некоторые форматы даты и времени по умолчанию задаются свойствами класса DateTimeFormatInfo (указаны в скобках после описания формата), при необходимости их можно менять.

1) «d» – короткая дата (ShortDatePattern). Пример:

```
      Console.WriteLine(ru.DateTimeFormat.ShortDatePattern);
      Console.WriteLine(v16.ToString("d", ru));
      Console.WriteLine(v16.ToString("d", us));
      Console.WriteLine(v16.ToString("d", DateTimeFormatInfo.InvariantInfo));
      Console.WriteLine(v16.ToString("d",
CultureInfo.CreateSpecificCulture("en-NZ")));
```

Вывод на консоль:

```
dd.MM.yyyy
20.08.2010
8/20/2010
08/20/2010
20/08/2010
```

В Новой Зеландии формат даты похож на российский, только в качестве разделителя используется слеш, а не точка.

2) «D» – полная дата (LongDatePattern). Пример:

```
      Console.WriteLine(us.DateTimeFormat.LongDatePattern);
      Console.WriteLine(v16.ToString("D", us));
      Console.WriteLine(v16.ToString("D",
CultureInfo.CreateSpecificCulture("pt-BR")));
      Console.WriteLine(v16.ToString("D",
CultureInfo.CreateSpecificCulture("es-MX")));
```

Вывод на консоль:

```
dddd, MMMM dd, yyyy
Friday, August 20, 2010
sexta-feira, 20 de agosto de 2010
viernes, 20 de agosto de 2010
```

Видим даты по стандартам американского диалекта английского языка, бразильского диалекта португальского и мексиканского диалекта испанского.

3) «f» – полная дата, короткое время. Пример:

```
      Console.WriteLine(v16.ToString("f", us));
      Console.WriteLine(v16.ToString("f",
CultureInfo.CreateSpecificCulture("fr-FR")));
```

Вывод на консоль:

```
Friday, August 20, 2010 6:30 PM
vendredi 20 aout 2010 18:30
```

Использован стандарт США и Франции.

4) «F» – полная дата и полное время (FullDateTimePattern). Пример:

```
      Console.WriteLine(ru.DateTimeFormat.FullDateTimePattern);
      Console.WriteLine(v16.ToString("F", ru));
      Console.WriteLine(v16.ToString("F",
CultureInfo.CreateSpecificCulture("fr-FR")));
```

Вывод на консоль:

```
d MMMM yyyy 'г.' H:mm:ss
20 августа 2010 г. 18:30:00
vendredi 20 aout 2010 18:30:00
```

5) «g» – короткая дата и короткое время. Пример:

```
      Console.WriteLine(v16.ToString("g", us));
```

```
      Console.WriteLine(v16.ToString("g", DateTimeFormatInfo.InvariantInfo));
      Console.WriteLine(v16.ToString("g",
CultureInfo.CreateSpecificCulture("fr-FR")));
```

Вывод на консоль:

```
      8/20/2010 6:30 PM
      08/20/2010 18:30
      20/08/2010 18:30
```

6) «G» – короткая дата, полное время. Пример:

```
      Console.WriteLine(v16.ToString("G", ru));
      Console.WriteLine(v16.ToString("G", DateTimeFormatInfo.InvariantInfo));
      Console.WriteLine(v16.ToString("G",
CultureInfo.CreateSpecificCulture("fr-FR")));
```

Вывод на консоль:

```
      20.08.2010 18:30:00
      08/20/2010 18:30:00
      20/08/2010 18:30:00
```

7) «M», «m» – месяц и день (MonthDayPattern). Пример:

```
      Console.WriteLine(ru.DateTimeFormat.MonthDayPattern);
      Console.WriteLine(v16.ToString("m", ru));
      Console.WriteLine(v16.ToString("m",
CultureInfo.CreateSpecificCulture("pt-BR")));
```

Вывод на консоль:

```
      MMMM dd
      августа 20
      20 de agosto
```

8) «t» – короткий шаблон времени (ShortTimePattern). Пример:

```
      Console.WriteLine(us.DateTimeFormat.ShortTimePattern);
      Console.WriteLine(v16.ToString("t", us));
      Console.WriteLine(v16.ToString("t",
CultureInfo.CreateSpecificCulture("es-ES")));
```

Вывод на консоль:

```
      h:mm tt
      6:30 PM
      18:30
```

9) «T» – полный шаблон времени (LongTimePattern). Пример:

```
      Console.WriteLine(ru.DateTimeFormat.LongTimePattern);
      Console.WriteLine(v16.ToString("T", us));
      Console.WriteLine(v16.ToString("T", ru));
```

Вывод на консоль:

```
      H:mm:ss
      6:30:00 PM
      18:30:00
```

10) «Y», «y» – месяц и год (YearMonthPattern). Пример:

```
      Console.WriteLine(us.DateTimeFormat.YearMonthPattern);
      Console.WriteLine(v16.ToString("y", us));
      Console.WriteLine(v16.ToString("y",
CultureInfo.CreateSpecificCulture("it-IT")));
```

Вывод на консоль:

```
      MMMM, yyyy
      August, 2010
      agosto 2010
```

Использованы стандарты США и Италии.

Существуют также и другие, реже используемые форматы:

• «O», «o» – шаблон цикла обработки даты и времени;

• «R», «r» – шаблон RFC 1123;

• «s» – сортируемый шаблон даты и времени;

• «U» – полный универсальный шаблон даты и времени.

Справку о них можно посмотреть в библиотеке MSDN. Для всех остальных спецификаторов формата даты и времени компилятор генерирует исключительную ситуацию FormatException.

Форматирование типов bool, char и string

В силу своей специфики данные типы практически не нуждаются в форматировании. Тип **bool** имеет всего два возможных значения, а типы **char** и **string** и так являются символьными.

Все эти типы наследуют метод ToString() от класса Object, а также метод ToString(IFormatProvider provider) от интерфейса IConvertible. Однако, экземпляр интерфейса provider для этих типов данных игнорируется, поэтому оба метода будут возвращать идентичные строки.

При форматировании булева типа данных метод ToString вернет значение константы Boolean.FalseString или Boolean.TrueString. При форматировании строки оба варианта метода вернут саму исходную строку. При форматировании символьного типа данных символ преобразуется в эквивалентное ему строковое представление. Также для символьного типа доступен еще один вариант метода:

```
      static string ToString(char c);
```

Делает он то же самое. В принципе, вызов **char**.ToString(c) эквивалентен вызову c.ToString().

3.2.1.4. Шаблоны пользователя

Шаблоны позволяют более полно задавать формат выдачи значений. По сути, задаются маски вывода. Если какая-либо часть шаблона заключена в одинарные или двойные кавычки, то она рассматривается просто как строка, а не как формат.

Шаблоны для чисел

1) «0» – заместитель нуля. Пример:

```
value = 123;
Console.WriteLine(value.ToString("00000"));
value = 1.2;
Console.WriteLine(value.ToString("0.00"));
Console.WriteLine(value.ToString("00.00"));
Console.WriteLine(value.ToString("00.00", us));
value = .56;
Console.WriteLine(value.ToString("0.0"));
```

Вывод на консоль:

```
00123
1,20
01,20
01.20
0,6
```

2) «#» – заместитель цифры. Пример:

```
value = 1.2;
Console.WriteLine(value.ToString("#.##"));
value = 123;
Console.WriteLine(value.ToString("#####"));
value = 123456;
Console.WriteLine(value.ToString("[##-##-##]"));
value = 1234567890;
Console.WriteLine(value.ToString("#"));
Console.WriteLine(value.ToString("(###) ###-####"));
value = 1234567890;
Console.WriteLine(value.ToString("#,#"));
Console.WriteLine(value.ToString("#,#", us));
value = 34.5;
Console.WriteLine(value.ToString("##"));
```

Вывод на консоль:

```
1,2
123
[12-34-56]
1234567890
(123) 456-7890
1 234 567 890
1,234,567,890
35
```

3) «.» – десятичный разделитель. Пример:

```
value = 1.2;
Console.WriteLine(value.ToString("0.00"));
Console.WriteLine(value.ToString("00.00"));
Console.WriteLine(value.ToString("00.00", us));
value = .086;
Console.WriteLine(value.ToString("#0.##%"));
value = 86000;
Console.WriteLine(value.ToString("0.###E+0"));
```

Вывод на консоль:

```
1,20
01,20
01.20
8,6%
8,6E+4
```

4) «,» – разделитель групп разрядов. Пример:

```
value = 1234567890;
Console.WriteLine(value.ToString("#,#"));
Console.WriteLine(value.ToString("#,##0,,"));
Console.WriteLine(value.ToString("#,,"));
Console.WriteLine(value.ToString("#,,,"));
```

Вывод на консоль:

```
1 234 567 890
1 235
1235
1
```

5) «%» – заместитель процентов. Пример:

```
value = .086;
Console.WriteLine(value.ToString("#0.##%"));
```

Вывод на консоль:

```
8,6%
```

6) «‰» – заместитель промилле. Пример:

```
value = .00354;
Console.WriteLine(value.ToString("#0.##‰"));
```

Вывод на консоль:

```
3.54‰
```

Примечание. Для вывода этого символа на консоль должна использоваться кодировка Unicode.

7) «E|e[+|–]0» – экспоненциальное представление чисел. Пример:

```
value = 86000;
Console.WriteLine(value.ToString("0.###E+0"));
```

```
Console.WriteLine(value.ToString("0.###E+000"));
Console.WriteLine(value.ToString("0.###E-000"));
```

Вывод на консоль:

```
8,6E+4
8,6E+004
86000,000E-000
```

8) «;» – разделитель секций. Если указаны две секции, то первая – для положительных чисел и нуля, вторая – для отрицательных чисел. Если три секции – первая для положительных чисел, вторая для отрицательных чисел, третья – для нуля. Пример:

```
value = 1234;
Console.WriteLine(value.ToString("##;(##)"));
value = -1234;
Console.WriteLine(value.ToString("##;(##)"));
value = 0;
Console.WriteLine(value.ToString("##;(##);ноль!"));
```

Вывод на консоль:

```
1234
(1234)
ноль!
```

Шаблоны для даты и времени

1) «d» – день месяца (1-31);

2) «dd» – день месяца (01-31);

3) «M» – месяц (1-12);

4) «MM» – месяц (01-12).

Пример:

```
Console.WriteLine(v16.ToString("d.M", ru));
Console.WriteLine(v16.ToString("dd.MM", ru));
```

Вывод на консоль:

```
20.8
20.08
```

5) «ddd» – сокращенное название дня недели (согласно элементам массива DateTimeFormatInfo.AbbreviatedDayNames);

6) «dddd…» – полное название дня недели (согласно элементам массива DateTimeFormatInfo.DayNames);

7) «MMM» – сокращенное название месяца (согласно элементам массива DateTimeFormatInfo.AbbreviatedMonthNames);

8) «MMMM…» – полное название месяца (согласно элементам массива DateTimeFormatInfo.MonthNames).

Пример:

```
      Console.WriteLine(v16.ToString("ddd d MMM", us));
      Console.WriteLine(v16.ToString("ddd d MMM",
CultureInfo.CreateSpecificCulture("fr-FR")));
      Console.WriteLine(v16.ToString("d MMMM",
CultureInfo.CreateSpecificCulture("es-MX")));
      Console.WriteLine(v16.ToString("dddd dd MMMM",
CultureInfo.CreateSpecificCulture("it-IT")));
      Console.WriteLine(v16.ToString("dddd dd MMMM", ru));
```

Вывод на консоль:

```
      Fri 20 Aug
      ven. 20 aout
      20 agosto
      venerdi 20 agosto
      пятница 20 августа
```

9) «y» – год (0-99);

10) «yy» – год (00-99);

11) «yyy» – год (000-…);

12) «yyyy» – год (0000-9999);

13) «yyyyy…» – год (00000-99999);

14) «g…» – эра.

Пример:

```
      Console.WriteLine(v16.ToString("MM/dd/yyy g", us));
      Console.WriteLine(v16.ToString("MM/dd/yyyy g",
CultureInfo.CreateSpecificCulture("fr-FR")));
```

Вывод на консоль:

```
      08/20/2010 A.D.
      08/20/2010 ap. J.-C.
```

15) «f», «ff», «fff», «ffff», «fffff», «ffffff», «fffffff» – доли секунды;

16) «F», «FF», «FFF», «FFFF», «FFFFF», «FFFFFF», «FFFFFFF» – доли секунды (незначащие нули не отображаются);

17) «s» – секунды (0-59);

18) «ss…» – секунды (00-59);

19) «m» – минуты (0-59);

20) «mm…» – минуты (00-59);

21) «h» – часы (1-12);

22) «hh…» – часы (01-12);

23) «H» – часы (0-23);

24) «HH…» – часы (00-23);

Пример:

```
Console.WriteLine(v16.ToString("hh:m:ss.f"));
Console.WriteLine(v16.ToString("hh:m:ss.F"));
Console.WriteLine(v16.ToString("H:mm:s.ff"));
Console.WriteLine(v16.ToString("H:mm:s.FF"));
Console.WriteLine(v16.ToString("HH:mm:ss.fff"));
Console.WriteLine(v16.ToString("HH:mm:ss.FFF"));
Console.WriteLine(v16.ToString("h:mm:ss.ffff"));
Console.WriteLine(v16.ToString("h:mm:ss.FFFF"));
```

Вывод на консоль:

```
06:30:00.0
06:30:00
18:30:0.05
18:30:0.05
18:30:00.055
18:30:00.055
6:30:00.0550
6:30:00.055
```

25) «t» – первый знак AM/PM (в соответствии со значениями **string** DateTimeFormatInfo.AMDesignator, **string** DateTimeFormatInfo.PMDesignator);

26) «tt…» – полная строка AM/PM;

27) «z», «zz», «zzz…» – смещение часового пояса.

Пример:

```
Console.WriteLine(v16.ToString("hh:mm t", us));
Console.WriteLine(v16.ToString("hh:mm tt zzz", us));
ru.DateTimeFormat.PMDesignator = "вечера";
Console.WriteLine(v16.ToString("hh:mm tt", ru));
```

Вывод на консоль:

```
06:30 P
06:30 PM +07:00
06:30 вечера
```

28) «:» – разделитель времени (в соответствии со значением **string** DateTimeFormatInfo.TimeSeparator);

29) «/» – разделитель даты (в соответствии со значением **string** DateTimeFormatInfo.DateSeparator).

Пример:

```
Console.WriteLine(v16.ToString("dd/MM/yyyy", ru));
Console.WriteLine(v16.ToString("dd'/'MM'/'yyyy", ru));
Console.WriteLine(v16.ToString("dd\"/\"MM\"/\"yyyy", ru));
Console.WriteLine(v16.ToString("\tt", us));
```

Вывод на консоль:

```
20.08.2010
20/08/2010
20/08/2010
        P
```

В первой дате в качестве разделителя использовался разделитель даты, а для русского языка это точка. Во втором случае слеш стоял в одинарных кавычках, поэтому выводился на консоль как обычный символ. В третьем случае то же самое, но кавычки двойные (ставим перед ними обратный слеш, чтобы кавычки считались символом строки, а не ее концом). В четвертом случае, несмотря на наличие в строке формата «tt», первая буква «t» входит в управляющий символ табуляции, поэтому на консоль будет выведена табуляция и время в формате «t».

Можно заметить, что некоторые шаблоны даты и времени совпадают с форматами. Например, формат «t» выводит короткое время, а шаблон «t» – первый знак AM/PM. Как компилятор их различает? Если в строке, кроме символа «t», других символов нет – это формат, если есть – шаблон.

3.2.1.5. Метод Format

Метод Format отличается от ToString тем, что позволяет форматировать сразу несколько объектов. Для этого в строку формата вводится новая конструкция:

```
"{"<номер аргумента>[,<ширина поля>][:<формат>]"}"
```

Здесь:

• Номер аргумента – номер форматируемого аргумента (нумерация начинается с нуля).

• Ширина поля – количество символов, которое отводится для строкового представления объекта. Если ширина положительная, используется выравнивание вправо, если отрицательная – влево.

• Формат – один из рассмотренных выше форматов (если формат не указан, используется формат по умолчанию).

Следовательно, в отличие от функции sprintf языка C++, один и тот же аргумент может быть получен в итоговой строке несколько раз в различных форматах, и порядок аргументов при вызове функции Format может отличаться от порядка вывода в итоговую строку.

Чтобы символы фигурных скобок в формате считались обычными символами, а не началом форматируемого поля, их нужно ставить парами.

Данный метод также позволяет осуществлять форматирование объектов типа **bool**, **char** и **string**. На них указанный формат не оказывает влияния.

Метод String.Format имеет два основных варианта синтаксиса:

```
      static string String.Format(string format, object obj0, ...);
      static string String.Format(IFormatProvider provider, string format,
object obj0, ...);
```

Отличаются они только тем, указывается или нет интерфейс управления форматированием. Далее идет строка формата и список форматируемых объектов.

Также в классе System.Enum есть метод Enum.Format. Он позволяет форматировать объекты перечисляемого типа. Можно использовать те же форматы, что и при форматировании перечислений методом ToString. Синтаксис метода:

```
      static string Enum.Format(Type enumType, object value, string format);
```

Пример:

```
      Console.WriteLine(String.Format("{0:t} {0:d}", v16));
      Console.WriteLine(String.Format("float: {0:e}, {0:f}, {0:g}\ndouble:
{1:e}, {1:f}, {1:g}", v9, v10));
      Console.WriteLine(String.Format("{0:dd/MM/yyyy}", v16));
      Console.WriteLine(String.Format(us, "{0:0.###E+0}", v9));
      Console.WriteLine(String.Format("{0,-10:g}!!!", v2));
      Console.WriteLine(String.Format("{0,10:g}!!!", v1));
      Console.WriteLine(String.Format("bool: {0,-10}!!!", v12));
      Console.WriteLine(String.Format("bool: {0,10}!!!", v12));
      Console.WriteLine(String.Format("char: {0,-10}!!!", v13));
      Console.WriteLine(String.Format("char: {0,10}!!!", v13));
      Console.WriteLine(String.Format("string: {0,-10}!!!", v14));
      Console.WriteLine(String.Format("string: {0,10}!!!", v14));
      Console.WriteLine(Enum.Format(v15.GetType(), v15, "g"));
      Console.WriteLine(Enum.Format(typeof(DayOfWeek), v15, "g"));
      Console.WriteLine(String.Format("{{a это не формат}}"));
```

Вывод на консоль:

```
      18:30 20.08.2010
      float: 2,478000e-001, 0,25, 0,2478
      double: 1,234568e+008, 123456789,12, 123456789,123457
      20.08.2010
      2.478E-1
      -111      !!!
             178!!!
      bool: True      !!!
      bool:       True!!!
      char: A         !!!
      char:          A!!!
      string: ABC       !!!
      string:        ABC!!!
      Friday
      Friday
```

```
      {а это не формат}
```

Как видно, в методе Format применимы все виды форматирования, доступные для методов ToString.

3.2.2. Вывод на консоль

Для вывода на консоль предназначены методы класса System.Console – Console.Write и Console.WriteLine. В классе Console есть и другие методы для управления выводом – копирования областей текста, изменения цвета текста и фона, управления курсором, смены кодировки и т.д. Все они для удобства пользователя являются статическими, т.е. чтобы не требовалось создавать экземпляры класса Console. Мы их рассматривать не будем, а полный список есть в библиотеке MSDN.

Синтаксис метода Write похож на синтаксис метода Format:

```
      static void Write(string format, object obj0, ...);
```

Есть и другие реализации этого метода, но все они работают одинаково. Если формат объекта не указан (в этом случае объект в списке аргументов может быть только один), например, Write(i), где i – целая переменная, то применяется формат по умолчанию. Т.е. эквивалентом такого вызова будет Write(i.ToString()) или Write("{0}", i).

Метод Write использует для форматирования метод String.Format, поэтому их возможности форматирования совпадают. Вся разница в том, что метод Write выводит отформатированную строку на консоль, а метод Format возвращает ее в качестве результата.

Поэтому результат работы следующего кода будет почти аналогичен результату работы кода для метода Format:

```
      Console.WriteLine("{0:t} {0:d}", v16);
      Console.WriteLine("float: {0:e}, {0:f}, {0:g}\ndouble: {1:e}, {1:f},
{1:g}", v9, v10);
      Console.WriteLine("{0:dd/MM/yyyy}", v16);
      Console.WriteLine("{0:0.###E+0}", v9);
      Console.WriteLine("{0,-10:g}!!!", v2);
      Console.WriteLine("{0,10:g}!!!", v1);
      Console.WriteLine("bool: {0,-10}!!!", v12);
      Console.WriteLine("bool: {0,10}!!!", v12);
      Console.WriteLine("char: {0,-10}!!!", v13);
      Console.WriteLine("char: {0,10}!!!", v13);
      Console.WriteLine("string: {0,-10}!!!", v14);
      Console.WriteLine("string: {0,10}!!!", v14);
      Console.WriteLine(v15);
      Console.WriteLine(v15);
      Console.WriteLine("{{а это не формат}}");
```

Результат работы:

```
18:30 20.08.2010
float: 2,478000e-001, 0,25, 0,2478
double: 1,234568e+008, 123456789,12, 123456789,123457
20.08.2010
2,478E-1
-111      !!!
       178!!!
bool: True      !!!
bool:       True!!!
char: A         !!!
char:          A!!!
string: ABC       !!!
string:        ABC!!!
Friday
Friday
{{a это не формат}}
```

Есть только два отличия:

1) Число «2,478E-1» при помощи метода Format выводилось в виде «2.478E-1», т.к. мы имели возможность указать реализацию интерфейса IFormatProvider, а для метода Write ее указать нельзя;

2) В последней строке парные скобки ставить не нужно, т.к. списка аргументов-объектов у функции Write нет, и строку «{a это не формат}» она воспринимает просто как строку. Если бы объекты были указаны, то скобки следовало бы сделать парными. Пример:

```
Console.WriteLine("{a это не формат}");
Console.WriteLine("{{a это не формат}}", v1);
Console.WriteLine("{a это не формат}", v1);
```

Результат работы:

```
{a это не формат}
{a это не формат}
```

Третья строка не выводится, а генерирует исключительную ситуацию System.FormatException, т.к. ожидается формат для переменной v1, но он указан неверно.

Метод WriteLine отличается от Write только тем, что выводит символ перевода строки в конце. Метод WriteLine без параметров просто переводит позицию вывода на следующую строку.

 Пример: Samples\3.2\3_2_2_write.

Напоследок упомянем об еще одном методе класса Console:

```
static void Clear();
```

Он удаляет из буфера консоли и ее окна отображаемую информацию

(аналог процедуры clrscr в языке C++ или консольной команды cls).

3.2.3. Методы Parse и TryParse

Все типы данных по значению (кроме структур) имеют методы Parse и TryParse, позволяющие преобразовать строку в значение данного типа. Т.е. это операция, обратная форматированию. В структуре DateTime определены четыре метода – Parse, TryParse, ParseExact и TryParseExact.

Разница между методами Parse и TryParse состоит в том, что первый из них возвращает в качестве результата считанное из строки значение, а если преобразование вызывает ошибку, генерирует исключительную ситуацию. Второй метод считанное из строки значение возвращает в виде выходного параметра (см. § 4.4), а возвращает признак успешного завершения разбора. Если разбор успешен, возвращается значение **true**, иначе **false**. Исключительные ситуации при этом не генерируются.

 Пример: Samples\3.2\3_2_3_parse.

3.2.3.1. Разбор строк с числовыми значениями

Все целые типы данных и типы данных с плавающей точкой имеют следующие реализации методов Parse и TryParse:

```
      static <тип> Parse(string s);
      static <тип> Parse(string s, NumberStyles style);
      static <тип> Parse(string s, IFormatProvider provider);
      static <тип> Parse(string s, NumberStyles style, IFormatProvider
provider);
      static bool TryParse(string s, out <тип> result);
      static bool TryParse(string s, NumberStyles style, IFormatProvider
provider, out <тип> result);
```

Здесь <тип> – один из типов **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double** и **decimal**. Ключевое слово **out** означает выходной параметр (см. § 4.4). Это аналог передачи параметра по ссылке в языке C++. Все методы являются статическими.

По аналогии с методами ToString, можно заключить, что наиболее универсальным является четвертый вариант метода Parse и второй вариант метода TryParse. Все остальные являются их упрощением – в них отсутствует либо ссылка на реализацию интерфейса IFormatProvider, либо стиль NumberStyles, либо и то, и другое.

Перечисление NumberStyles позволяет задать некоторые дополнитель-

ные опции преобразования (табл. 3.14).

Табл. 3.14 – Константы перечисления NumberStyles

Константа	Описание
`None (0x0000)`	Не разрешен ни один стиль
`AllowLeadingWhite (0x0001)`	Начальные пробелы не следует обрабатывать в процессе разбора. Допустимые знаки пробела имеют следующие значения в кодировке Unicode: U+0009, U+000A, U+000B, U+000C, U+000D и U+0020
`AllowTrailingWhite (0x0002)`	Конечные пробелы не следует обрабатывать в процессе разбора
`AllowLeadingSign (0x0004)`	Числовая строка может включать начальный знак (PositiveSign и NegativeSign класса NumberFormatInfo)
`AllowTrailingSign (0x0008)`	Числовая строка может включать конечный знак
`AllowParentheses (0x0010)`	В числовой строке может находиться одна пара скобок, в которых заключено число
`AllowDecimalPoint (0x0020)`	Числовая строка может включать десятичный разделитель (см. свойства NumberDecimalSeparator, CurrencyDecimalSeparator и PercentDecimalSeparator класса NumberFormatInfo)
`AllowThousands (0x0040)`	В числовой строке могут находиться разделители групп (NumberGroupSeparator, NumberGroupSizes, CurrencyGroupSeparator и CurrencyGroupSizes класса NumberFormatInfo)
`AllowExponent (0x0080)`	Числовая строка может быть в экспоненциальном представлении
`AllowCurrencySymbol (0x0100)`	Числовая строка разбирается как денежная единица, если она содержит символ денежной единицы (CurrencySymbol класса NumberFormatInfo)
`AllowHexSpecifier (0x0200)`	Числовая строка представляет шестнадцатеричное значение
`Integer (0x0007)`	Используется комбинация стилей AllowLeadingWhite, AllowTrailingWhite и AllowLeadingSign
`HexNumber (0x0203)`	Используется комбинация стилей AllowLeading-

	White, AllowTrailingWhite и AllowHexSpecifier
Number (0x006f)	Используется комбинация стилей AllowLeading-White, AllowTrailingWhite, AllowLeadingSign, Allow-TrailingSign, AllowDecimalPoint и AllowThousands
Float (0x00a7)	Используется комбинация стилей AllowLeading-White, AllowTrailingWhite, AllowLeadingSign, Allow-Exponent и AllowDecimalPoint
Currency (0x017f)	Используется комбинация всех стилей, за исключением AllowExponent и AllowHexSpecifier
Any (0x01ff)	Используется комбинация всех стилей, за исключением AllowHexSpecifier

Пример:

```
      string str;
      int v1;
      double v2;
      NumberFormatInfo nfi = new NumberFormatInfo();
      CultureInfo ru = CultureInfo.CreateSpecificCulture("ru-RU");
      CultureInfo us = CultureInfo.CreateSpecificCulture("en-US");

      str = "A";
      v1 = int.Parse(str, NumberStyles.HexNumber);
      Console.WriteLine("'0x{0}' = '{1}'",  str, v1);
      str = "    -45   ";
      v1 = int.Parse(str, NumberStyles.AllowLeadingSign |
            NumberStyles.AllowLeadingWhite |
            NumberStyles.AllowTrailingWhite);
      Console.WriteLine("'{0}' = '{1}'", str, v1);
      str = "  (37)  ";
      v1 = int.Parse(str, NumberStyles.AllowParentheses |
            NumberStyles.AllowLeadingSign |
            NumberStyles.AllowLeadingWhite |
            NumberStyles.AllowTrailingWhite);
      Console.WriteLine("'{0}' = '{1}'", str, v1);
      str = "123,456";
      v1 = int.Parse(str, NumberStyles.AllowThousands, us);
      Console.WriteLine("'{0}' = '{1}' ({2})", str, v1, us.Name);
      v2 = double.Parse(str, NumberStyles.AllowDecimalPoint, ru);
      Console.WriteLine("'{0}' = '{1}' ({2})", str, v2, ru.Name);

      if (int.TryParse(str, NumberStyles.AllowThousands, ru, out v1))
      {
            Console.WriteLine("'{0}' = '{1}'", str, v1);
      }
      else
      {
            Console.WriteLine("'{0}': не удалось преобразовать в тип int
({1})", str, ru.Name);
      }
```

Вывод на консоль:

```
'0xA' = '10'
'    -45   ' = '-45'
'  (37)  ' = '-37'
'123,456' = '123456' (en-US)
'123,456' = '123,456' (ru-RU)
'123,456': не удалось преобразовать в тип int (ru-RU)
```

3.2.3.2. Разбор строк со значениями bool и char

Для этих типов данных предусмотрены только два метода разбора:

```
static <тип> Parse(string value);
static bool TryParse(string value, out <тип> result);
```

Для логического типа разбор будет успешен, если строка содержит значение Boolean.FalseString или Boolean.TrueString (регистр символов не важен). Для символьного типа – если строка содержит один символ Unicode.

Пример:

```
bool v3;
char v4;

str = "true";
v3 = bool.Parse(str);
Console.WriteLine("'{0}' = '{1}'", str, v3);
str = "истина";

if (bool.TryParse(str, out v3))
{
      Console.WriteLine("'{0}' = '{1}'", str, v3);
}
else
{
      Console.WriteLine("'{0}': не удалось преобразовать в тип bool",
str);
}

str = "Ё";
v4 = char.Parse(str);
Console.WriteLine("'{0}' = '{1}'", str, v4);
str = "!!!";

if (char.TryParse(str, out v4))
{
      Console.WriteLine("'{0}' = '{1}'", str, v4);
}
else
{
      Console.WriteLine("'{0}': не удалось преобразовать в тип char",
str);
}
```

Вывод на консоль:

```
'true' = 'True'
'истина': не удалось преобразовать в тип bool
```

```
'Ё' = 'Ё'
'!!!': не удалось преобразовать в тип char
```

3.2.3.3. Разбор строк со значениями перечисляемого типа

В классе System.Enum есть два варианта метода Parse:

```
static object Parse(Type enumType, string value);
static object Parse(Type enumType, string value, bool ignoreCase);
```

Метод TryParse не предусмотрен. Параметр ignoreCase позволяет игнорировать регистр символов. Если искомая символическая константа в указанном перечислении не найдена, генерируется исключительная ситуация System.ArgumentException.

Пример:

```
DayOfWeek v5;

Console.WriteLine("Разбор перечисляемого типа");
str = "Friday";
v5 = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), str);
Console.WriteLine("'{0}' = '{1}'", str, v5);
str = "MonDAY";
v5 = (DayOfWeek)Enum.Parse(v5.GetType(), str, true);
Console.WriteLine("'{0}' = '{1}'", str, v5);
str = "zzz";

try
{
      v5 = (DayOfWeek)Enum.Parse(v5.GetType(), str, true);
      Console.WriteLine("'{0}' = '{1}'", str, v5);
}
catch (ArgumentException e)
{
      Console.WriteLine(e);
}
```

Вывод на консоль:

```
'Friday' = 'Friday'
'MonDAY' = 'Monday'
System.ArgumentException: Запрошенное значение "zzz" не найдено
```

3.2.3.4. Разбор строк с датой и временем

Для разбора даты и времени предусмотрены следующие методы:

```
static DateTime Parse(string s);
static DateTime Parse(string s, IFormatProvider provider);
static DateTime Parse(string s, IFormatProvider provider, DateTimeStyles
styles);
static DateTime ParseExact(string s, string format, IFormatProvider
provider);
static DateTime ParseExact(string s, string format, IFormatProvider
```

```
provider, DateTimeStyles style);
      static DateTime ParseExact(string s, string[] formats, IFormatProvider
provider, DateTimeStyles style);
      static bool TryParse(string s, out DateTime result);
      static bool TryParse(string s, IFormatProvider provider, DateTimeStyles
styles, out DateTime result);
      static bool TryParseExact(string s, string format, IFormatProvider
provider, DateTimeStyles style, out DateTime result);
      static bool TryParseExact(string s, string[] formats, IFormatProvider
provider, DateTimeStyles style, out DateTime result);
```

Методы Parse пытаются преобразовать строку к значению даты и времени, основываясь на указанной реализации интерфейса IFormatProvider и дополнительных опций, задаваемых перечислением DateTimeStyles (табл. 3.15).

Табл. 3.15 – Константы перечисления DateTimeStyles

Константа	Описание
None (0x0000)	Используются параметры форматирования по умолчанию
AllowLeadingWhite (0x0001)	Начальные знаки-разделители не следует учитывать при разборе
AllowTrailingWhite (0x0002)	Конечные знаки-разделители не следует учитывать при разборе
AllowInnerWhite (0x0004)	Лишние знаки-разделители в середине строки не следует учитывать при разборе
AllowWhiteSpaces (0x0007)	Лишние знаки-разделители в любом месте строки не следует учитывать при разборе (комбинация AllowLeadingWhite, AllowTrailingWhite и AllowInnerWhite)
NoCurrentDateDefault (0x0008)	Если в разбираемой строке содержится только время и отсутствует дата, подразумевается дата 1.1.1. Если это значение не используется, подразумевается текущая дата
AdjustToUniversal (0x0010)	Дата и время возвращаются в формате универсального времени (UTC)
AssumeLocal (0x0020)	Если часовой пояс не указан в строке разбора, подразумевается, что используется локальное время
AssumeUniversal	Если часовой пояс не указан в строке разбора,

(0x0040)	подразумевается, что используется формат UTC
RoundtripKind (0x0080)	Поле даты DateTimeKind сохраняется при преобразовании в строку с помощью стандартного описателя формата «o» или «r» и преобразования строки обратно

Методы ParseExact делают то же самое, но требуют точного совпадения строки даты и времени с указанным форматом или массивом форматов. Методы TryParse и TryParseExact отличаются от Parse и ParseExact тем же – при ошибке преобразования не выдают исключительную ситуацию, а возвращают **false**.

Пример:

```
DateTime v6;

Console.WriteLine("Разбор даты и времени:");
str = " Friday, August 20, 2010 18:30 ";
v6 = DateTime.Parse(str);
Console.WriteLine("'{0}' = '{1}'", str, v6);

try
{
      v6 = DateTime.ParseExact(str, "D", us);
      Console.WriteLine("'{0}' = '{1}'", str, v6);
}
catch (FormatException)
{
      Console.WriteLine("'{0}' не является датой в формате '{1}'",
            str, us.DateTimeFormat.LongDatePattern);
}

str = " Friday, August 20, 2010  ";
v6 = DateTime.ParseExact(str, "D", us, DateTimeStyles.AllowWhiteSpaces);
Console.WriteLine("'{0}' = '{1}'", str, v6);
```

Вывод на консоль:

```
      ' Friday, August 20, 2010 18:30 ' = '20.08.2010 18:30:00'
      ' Friday, August 20, 2010 18:30 ' не является датой в формате 'dddd,
MMMM dd, yyyy'
      ' Friday, August 20, 2010  ' = '20.08.2010 0:00:00'
```

Во втором случае преобразование вызвало ошибку потому, что в строке с датой были лишние пробелы, и время в формате «D» не должно содержаться. В третьем случае время из строки мы убрали, а лишние пробелы игнорируются, т.к. указан флаг AllowWhiteSpaces.

3.2.4. Ввод с консоли

Для чтения данных с консоли в классе System.Console предназначены следующие методы:

```
static int Read();
static string ReadLine();
static ConsoleKeyInfo ReadKey();
static ConsoleKeyInfo ReadKey(bool intercept);
```

 Пример: Samples\3.2\3_2_4_read.

3.2.4.1. Метод Read

Метод Read читает один символ из потока ввода. Он возвращает значение типа **int**, равное коду прочитанного символа, либо –1, если в буфере ввода нет символов. Может показаться, что этот метод похож на функцию getch из консольной библиотеки языка C++ (conio.h). Однако, это не так. Когда выполнение программы доходит до вызова метода Read, пользователь может ввести не один символ, а сразу целую строку, завершив ее ввод нажатием клавиши «Enter». Метод Read вернет первый символ введенной строки, следующий вызов данного метода вернет второй символ, и т.д.

Что интересно, в режиме чтения действуют некоторые функции командной строки Windows. Так, нажатие клавиш «↑» и «↓» будет помещать в позицию ввода строки, расположенные в истории ввода ранее или позже, и т.д.

Чтобы метод Read вернул –1, необходимо на консоли ввести символ окончания ввода (комбинация клавиш Ctrl+Z или F6, затем Enter).

Приведем пример программы:

```
int i;
string str;
char c;

Console.WriteLine("Вводите любые строки текста.");
Console.WriteLine("Можно использовать клавиши 'Вверх'");
Console.WriteLine("и 'Вниз' для перемещения по истории");
Console.WriteLine("ввода. Для выхода из режима ввода");
Console.WriteLine("нажмите Ctrl+Z или F6, затем Enter");

do
{
      i = Console.Read();

      if (i != -1)
      {
```

```
                c = Convert.ToChar(i);
                Console.Write("Символ с кодом U+{0:X4}", i);
                if (Char.IsControl(c)) Console.WriteLine(" [управляющий]");
                else Console.WriteLine(" = '{1}'", i, c);
        }
    } while (i != -1);
```

Результаты работы программы:

```
привет!
Символ с кодом U+043F = 'п'
Символ с кодом U+0440 = 'р'
Символ с кодом U+0438 = 'и'
Символ с кодом U+0432 = 'в'
Символ с кодом U+0435 = 'е'
Символ с кодом U+0442 = 'т'
Символ с кодом U+0021 = '!'
Символ с кодом U+000D [управляющий]
Символ с кодом U+000A [управляющий]
^Z
```

В данном примере сначала была введена строка «привет!». В буфер клавиатуры попали символы этой строки, а также символы «\r» (0x0d) и «\n» (0x0a), которыми завершился ее ввод. Затем информация об этих символах была выведена на консоль. При следующем запросе на ввод строки была нажата комбинация клавиш Ctrl+Z и Enter, завершившая ввод.

3.2.4.2. Метод ReadLine

Метод ReadLine читает из потока ввода строку текста, которая завершается символом перевода строки и/или возврата каретки («\r» и/или «\n»). Метод возвращает объект типа **string** или **null**, если ввод осуществить не удалось (т.е. если в буфере клавиатуры или другого потока ввода был символ окончания ввода):

```
Console.WriteLine("Вводите любые строки текста.");
Console.WriteLine("Для выхода из режима ввода нажмите");
Console.WriteLine("Ctrl+Z или F6, затем Enter");

do
{
      str = Console.ReadLine();
      if (str != null)
      {
            Console.WriteLine("Введена строка '{0}'", str);
      }
} while (str != null);
```

Результаты работы программы:

```
привет!
Введена строка 'привет!'
^Z
```

После считывания строки со значениями можно осуществлять ее разбор различными модификациями метода Parse.

3.2.4.3. Метод ReadKey

Метод ReadKey, как и метод Read, позволяет прочитать из буфера клавиатуры или потока ввода один символ, но работает уже как аналог функции getch. Т.е. можно считывать не только обычные символы, но и сочетания клавиш.

Метод имеет две разновидности. Вторая позволяет управлять отображением введенного символа (если параметр intercept равен **true**, символ не отображается, иначе отображается на консоли). Первый вариант метода всегда отображает введенный символ.

Информация о нажатых клавишах возвращается в виде экземпляров структуры ConsoleKeyInfo (табл. 3.16).

Табл. 3.16 – Основные члены структуры ConsoleKeyInfo

Член	Описание
Конструкторы	
`ConsoleKeyInfo(char keyChar, ConsoleKey key, bool shift, bool alt, bool control)`	Инициализирует новый экземпляр структуры ConsoleKeyInfo с использованием заданного символа, клавиши консоли и управляющих клавиш
Операторы	
`static bool operator == (ConsoleKeyInfo a, ConsoleKeyInfo b)`	Указывает, равны ли заданные объекты
`static bool operator != (ConsoleKeyInfo a, ConsoleKeyInfo b)`	Указывает, являются ли заданные объекты неравными
Свойства	
`ConsoleKey Key`	Возвращает клавишу консоли, представленную текущим объектом
`char KeyChar`	Возвращает символ Unicode, представленный текущим объектом
`ConsoleModifiers Modifiers`	Возвращает состояние управляющих клавиши (Shift, Alt или Ctrl) – нажаты

	они или нет

Перечисление ConsoleModifiers содержит именованные константы Alt, Shift и Control. Левые и правые управляющие клавиши не различаются. Свойство KeyChar позволяет получить доступ к коду нажатой клавиши, если это обычный символ. Свойство Key – если была нажата любая клавиша клавиатуры. Перечисление ConsoleKey содержит именованные константы для каждой клавиши. Их несколько десятков, полный список можно посмотреть в библиотеке MSDN.

Как мы помним, в библиотеках языка C++ также была функция kbhit(), позволяющая узнать, есть ли во входном потоке данные о нажатых клавишах. В языке C# ее аналогом является следующее свойство класса Console:

```
static bool KeyAvailable;
```

Пример:

```
Console.WriteLine("Нажимайте любые комбинации клавиш,");
Console.WriteLine("чтобы увидеть их интерпретацию.);
Console.WriteLine("Выход по Ctrl+Пробел");

ConsoleKeyInfo exit = new ConsoleKeyInfo(' ', ConsoleKey.Spacebar,
false, false, true);
ConsoleKeyInfo key;

do
{
      key = Console.ReadKey(true);
      Console.Write("Вы нажали ");

      if (((int)key.Modifiers & (int)ConsoleModifiers.Alt) != 0)
            Console.Write("Alt+");
      if (((int)key.Modifiers & (int)ConsoleModifiers.Shift) != 0)
            Console.Write("Shift+");
      if (((int)key.Modifiers & (int) ConsoleModifiers.Control) != 0)
            Console.Write("Ctrl+");
      Console.WriteLine(Enum.GetName(typeof(ConsoleKey), key.Key));
} while (key != exit);
```

Результаты работы программы:

```
Вы нажали Shift+Ctrl+F
Вы нажали Alt+F5
Вы нажали Shift+Z
Вы нажали R
Вы нажали Shift+Ctrl+Insert
Вы нажали Ctrl+Spacebar
```

Результаты работы зависят от того, какие клавиши будут нажаты.

§ 3.3. Вычисление выражений

В данном параграфе мы рассмотрим вычисление выражений – это одна из трех основ структурного программирования. Две другие – организация в программе ветвлений и циклов, рассмотрены в следующем параграфе. Вычисление выражений состоит в применении операций языка по отношению к данным (аргументам операций).

При вычислении выражений используются *операторы выражений*. Также существуют *операторы языка* C#, рассмотренные в § 3.4. В дальнейшем мы как операторы выражений, так и операторы языка будем называть просто операторами. О том, какие именно операторы имеются в виду, будет ясно из контекста.

Итак, оператор выражения – это символ, указывающий, какую операцию необходимо выполнить над одним или несколькими аргументами. При выполнении оператора получается результат. Синтаксис применения операторов несколько отличен от вызова методов, поэтому необходимо знать формат выражений, содержащих операторы языка C#. Как и в большинстве других языков, семантика операторов в C# соответствует правилам и нотациям, знакомым нам из математики и из языка C++.

3.3.1. Набор операторов языка C#

Стандартные операторы, определенные в языке C#, перечислены в табл. 3.17.

Табл. 3.17 – Операторы языка C# (по категориям)

Категория	Операторы
Арифметические	`+ - * / %`
Инкремента и декремента	`++ --`
Логические	`&& || ! true false`
Побитовые	`& | ^ ~ << >>`
Сравнения	`== != < > <= >=`
Присваивания	`= *= /= %= += -= <<= >>= &= ^= |=`
Доступа	`. ::`
Индексирования	`[]`
Преобразования типа	`() as`

Выбора	`?: ??`
Создания объектов	`new stackalloc`
Рефлективные	`sizeof is typeof`
Управления переполнением	`checked unchecked`
Работа с указателями	`* -> & []`
Лямбда-выражение	`=>`

Данное деление на категории несколько условно. Например, арифметический оператор «+» также применяется для конкатенации строк. Оператор «()» используется не только для преобразования типов, но и для вызова методов, а также как обычные скобки в выражении, указывающие порядок выполнения операций. А оператор «->» можно отнести как к операторам работы с указателями, так и к операторам доступа. И т.д.

3.3.2. Приоритет и порядок выполнения

Когда в одном выражении несколько операторов, компилятор должен определить порядок их выполнения. При этом компилятор руководствуется правилами, которые называются *старшинством операторов*. Понимание старшинства операторов нужно для правильного написания выражений – иногда результат может не соответствовать ожидаемому.

Посмотрим, как старшинство операторов определяется в C# (табл. 3.18).

Табл. 3.18 – Операторы языка C# (по приоритету)

Приоритет	Группа	Операции	Порядок
0	Первичные	`()`(1) `.` `::` `()`(2) `[]` `++`(3) `--`(3) `->`(4) `new` `typeof` `checked` `unchecked` `default` `delegate` `stackalloc`(4)	→
1	Унарные	`+` `-` `!` `~` `++`(5) `--`(5) `()`(6) `&`(4, 7) `*`(4, 8) `sizeof`(4) `true` `false`	→
2	Мультипликативные	`*` `/` `%`	→
3	Аддитивные	`+` `-`	→
4	Сдвига	`<<` `>>`	→

5	Отношения и проверки типов	`< > <= >= is as`	→
6	Эквивалентности	`== !=`	→
7	Побитовые	`&`	→
8		`^`	→
9		`|`	→
10	Булевы	`&&`	→
11		`||`	→
12	Поддержка **null**	`??`	←
13	Условный	`?:`	←
14	Присваивания	`= *= /= %= += -= <<= >>= &= ^= |=`	←
	Лямбда-выражение	`=>`	←

Примечания:

(1) скобки;

(2) оператор вызова функции;

(3) постфиксные;

(4) только в небезопасном коде;

(5) префиксные;

(6) преобразование типа;

(7) получение адреса;

(8) разыменование указателя.

При конструировании выражений важно учитывать приоритет операций. Например, результат выражения

```
false && false || true
```

зависит от того, у какого оператора выше приоритет – логической конъюнкции или дизъюнкции. Если у конъюнкции, то получим

```
(false && false) || true = false || true = true
```

А если у дизъюнкции, то

```
false && (false || true) = false && true = false
```

Если нет уверенности в очередности выполнения операций, то лучше задать ее явно, используя скобки. Важно, что у операторов присваивания самый низкий приоритет (14). Иначе в записи

```
<переменная> = <выражение>
```

выражение практически всегда приходилось бы брать в скобки.

По порядку выполнения операторы разделяются на операторы с левой и правой ассоциативностью. Ассоциативность определяет, какая часть выражения должна быть вычислена первой. Например, результатом приведенного выражения может быть 21 или 33 в зависимости от того, какая ассоциативность будет применяться для оператора «–»: левая или правая:

```
42 - 15 - 6
```

Данный оператор имеет левую ассоциативность, т.е. сначала вычисляется 42–15, а затем из результата вычитается 6, результат – 21. Если бы он имел правую ассоциативность, сначала вычислялась бы правая часть выражения (15–6 = 9), а затем результат вычитался бы из: 42 – 9 = 33.

Все бинарные операторы (операторы с двумя операндами), кроме операторов присваивания, – лево-ассоциативные, т.е. они обрабатывают выражения слева направо. Таким образом, «a + b + c» – то же, что и «(a + b) + c», где сначала вычисляется «a + b», а затем к сумме прибавляется «c». Операторы присваивания и условные операторы – право-ассоциативные, т.е. обрабатывают выражения справа налево. Иначе говоря, «a = b = c» эквивалентно «a = (b = c)». Например, рассмотрим следующий код:

```
int a = 1;
int b = 2;
int c = 3;
a = b = c;
```

В результате все три переменные будут иметь значение 3. Если бы оператор присваивания был лево-ассоциативным, то сначала выполнился бы оператор «a = b», а затем – «b = c». В итоге переменная «a» имела бы значение 2, а переменные «b» и «c» – 3. Очевидно, что мы ожидаем не этого, когда пишем «a = b = c», и именно поэтому операторы присваивания и условные операторы право-ассоциативные.

3.3.3. Описание операторов

Рассмотрим перечисленные выше операторы по категориям. Но сначала выделим некоторые первичные операторы, которые нам уже знакомы или которые будут изучены позже.

 Пример: Samples\3.3\3_3_3_oper.

3.3.3.1. Первичные операторы

Рассмотрим сначала некоторые простые первичные операторы, с которыми мы уже сталкивались ранее или рассмотрим в последующих разделах:

• (x) – Это разновидность оператора «скобки» для управления порядком вычислений в математических, логических и других операциях;

• x.y – Оператор «точка» используется для указания члена пространства имен, класса, структуры или интерфейса. Здесь «x» представляет сущность, содержащую в себе член «y»;

• x::y – Квалификатор псевдонима пространства имен. Здесь «x» представляет псевдоним пространства имен, содержащего в себе член «y» (§ 4.1);

• f(x) – Такая разновидность оператора «скобки» применяется для вызова методов;

• a[x] – Квадратные скобки используются для индексации массива;

• **new** – Этот оператор используется для создания экземпляров объектов (классов, структур, делегатов и т.д.);

• **stackalloc** – Выделение блока памяти на стеке (п. 5.5.2.4);

• **default** – Возвращает значение типа по умолчанию (§ 3.1);

• **delegate** – Оператор объявления анонимной функции (§ 4.8).

Остальные первичные операторы рассмотрены далее, по категориям. Кроме условного тернарного оператора, остальные операторы являются унарными или бинарными (оператора). Синтаксис операторов будем записывать в следующей форме:

1) Для префиксных или постфиксных унарных операторов:

```
<тип результата> operator op (<тип аргумента> x)
```

В префиксных операторах знак операции (op) стоит перед аргументом (op x), в постфиксных операциях – после аргумента (x op).

2) Для инфиксных бинарных операторов:

```
<тип результата> operator op (<тип аргумента1> x, <тип аргумента2> y)
```

В инфиксных операторах знак операции (op) стоит между аргументами (x op y).

Если оператор применим для какого-либо типа по значению <тип>, то он применим и для соответствующего обнуляемого типа <тип>?. При этом, если хотя бы один аргумент имеет неопределенное значение (**null**), то и результатом также будет **null.** Исключения будут оговариваться отдельно.

3.3.3.2. Арифметические операторы

Язык C#, как и большинство других языков, поддерживает основные математические операторы – бинарные (умножение, деление, сложение, вычитание и модуль) и унарные (плюс и минус). Синтаксис бинарных операторов (*, /, +, –, %):

```
<тип> operator op (<тип> x, <тип> y)
```

Здесь <тип> – один из типов **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**. Операторы «+» и «–» также определены для делегатов (см. § 4.8). Для выполнения арифметических операций с другими типами данных используется неявное приведение типов. А если оно не определено, компилятор генерирует ошибку:

```
byte b = 1;
b = b + b; // Ошибка
```

В данном примере используется оператор сложения с аргументами типа **int**, т.к. это ближайший тип, перекрывающий диапазон типа **byte**. Полученный результат также имеет тип **int**, но нет неявного преобразования из **int** в **byte**, поэтому компилятор покажет ошибку.

Унарных арифметических операторов четыре: плюс, минус, инкремент и декремент. Первые два являются префиксными, последние два имеют префиксную и постфиксную форму.

Оператор унарного минуса указывает компилятору, что результатом операции будет значение аргумента с обратным знаком. Синтаксис:

```
<тип> operator - (<тип> x)
```

Допустимые типы – **int**, **long**, **float**, **double**, **decimal**. В операторе унарного плюса результатом будет просто значение аргумента. Синтаксис:

```
<тип> operator + (<тип> x)
```

Допустимые типы – **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**.

Операторы инкремента (++) и декремента (--) позволяют лаконично выразить, что мы хотим увеличить или уменьшить числовое значение на 1. В префиксной версии операторов сначала выполняется операция, а затем создается значение. В постфиксной версии сначала создается значение, а затем выполняется операция. Синтаксис операторов:

```
<тип> operator ++ (<тип> x)
<тип> operator -- (<тип> x)
```

Допустимые типы – **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, **double**, **decimal** и перечисления. При этом аргумент «x» обязательно должен попадать в категорию l-value (см. п. 3.3.3.6).

Пример:

```
int v1 = 1;
uint v2 = 2;
byte v3 = 1;
sbyte v4 = 2;
long v5 = 1;
ulong v6 = 2;
double v7 = 1;
float v8 = 2;
decimal v9 = 3;
char v10 = 'A';
bool v11 = false;

Console.WriteLine("int + uint = " + (v1 + v2).GetType());
Console.WriteLine("byte - sbyte = " + (v3 - v4).GetType());
Console.WriteLine("uint * ulong = " + (v2 * v6).GetType());
Console.WriteLine("uint / byte = " + (v2 / v3).GetType());
Console.WriteLine("byte % byte = " + (v3 % v3).GetType());
Console.WriteLine("long + ulong = " + (v5 + v6).GetType()); // Ошибка
Console.WriteLine("double * int = " + (v7 * v1).GetType());
Console.WriteLine("double - decimal = " + (v7 – v9).GetType()); //
Ошибка
Console.WriteLine("float - float = " + (v8 - v8).GetType());
Console.WriteLine("char * char = " + (v10 * v10).GetType());
Console.WriteLine("bool * bool = " + (v11 * v11).GetType()); // Ошибка

DayOfWeek v12 = DayOfWeek.Friday;
int a1 = 1;
int b1 = a1++;
int a2 = 1;
int b2 = ++a2;

Console.WriteLine("{0} {1}", a1, b1);
Console.WriteLine("{0} {1}", a2, b2);

int b3 = 1++; // Ошибка

Console.WriteLine(++v10);
Console.WriteLine(++v12);
```

Первая ошибка связана с тем, что нет типа данных с диапазоном значений, перекрывающим диапазоны значений типов **long** и **ulong**. Вторая – с тем, что не определены неявные преобразования между типами **double** и **decimal**. Третья – с тем, что тип **bool** нельзя неявно преобразовать ни к одному из типов, для которых определены арифметические операторы. Четвертая – аргумент инкремента не является l-value. Результаты работы программы:

```
int + uint = System.Int64
byte - sbyte = System.Int32
uint * ulong = System.UInt64
```

```
uint / byte = System.UInt32
byte + byte = System.Int32
double * int = System.Double
float - float = System.Single
char * char = System.Int32
2 1
2 2
B
Saturday
```

3.3.3.3. Логические операторы

К логическим операторам относятся условные логические операторы конъюнкция (`&&`) и дизъюнкция (`||`), а также логические операторы отрицания (`!`) и пользовательские условные логические операторы **true** и **false**.

Условные логические операторы являются бинарными:

```
bool operator && (bool x, bool y)
bool operator || (bool x, bool y)
```

Данные операторы применимы только к аргументам логического типа, а также к типам, определяющим преобразование к типу **bool** (как определить операции преобразования типа, читайте в § 4.7). Для обнуляемого типа **bool**? они не применимы.

Операторы «`&&`» и «`||`» являются условными версиями операторов «`&`» и «`|`» для типа **bool**:

• Операция «`x && y`» соответствует операции «`x & y`», за исключением того, что «y» вычисляется, только если «x» не равен **false**.

• Операция «`x || y`» соответствует операции «`x | y`», за исключением того, что «y» вычисляется, только если «x» не равен **true**.

Синтаксис унарных логических операторов:

```
bool operator ! (bool x)
bool operator true (<тип> x)
bool operator false (<тип> x)
```

Первый оператор вычисляет логическое отрицание операнда. Операторы **true** и **false** определяются в классе пользователя для того, чтобы экземпляры класса могли:

1) использоваться в качестве условия в операторах **if**, **for**, **while**, а также в условном операторе (?:) (см. пример);

2) являться аргументами логических операторов (см. § 4.7).

Пример:

```
CInt v16 = new CInt(1);
```

```
v1 = 1;
if (v1 && v11) Console.WriteLine(v1); // Ошибка
v11 = false;
if (v11 && (v1++) != 0) Console.WriteLine(v1);
v11 = true;
if (v11 || (v1++) != 0) Console.WriteLine(v1);
if (true || (v1++) != 0) Console.WriteLine(v1); // Предупреждение
Console.WriteLine(v1);
if (v16) Console.WriteLine(v16.Value + " == true");
```

Вывод на консоль:

```
1
1
1
1 == true
```

Видно, что код «v1++» ни разу не был выполнен (данное выражение не обязательно заключать в скобки, т.к. приоритет инкремента выше, чем у операций сравнения; это сделано для повышения читабельности кода). В первом случае потому, что «false && x» всегда равно **false**, поэтому вычисление аргумента «x» не требуется. Во втором случае – потому что «true || x» всегда равно **true**, и вычисление аргумента «x» также не требуется. В третьем случае в логическом операторе использована не переменная, а константа, поэтому код «v1++» не будет выполнен ни при каких условиях. Компилятор генерирует предупреждение «Обнаружен недостижимый код в выражении».

Ошибка связана с тем, что оператор «int && bool» не определен, и нет неявного преобразования типа **int** в тип **bool**.

В последней строке переменная v16 является аргументом условия. Это возможно, т.к. класс CInt перегружает операторы **true** и **false** (см. исходный код решения к данному пункту). Видно, что данные операторы вызываются неявно. Запись **true**(v16) или **false**(v16) привела бы к ошибке компиляции.

3.3.3.4. Побитовые операторы

К побитовым операторам относятся побитовые логические операторы конъюнкция (&), дизъюнкция (|), исключающая дизъюнкция (^) и дополнение (~), а также операторы побитового сдвига (<< и >>).

Синтаксис бинарных побитовых логических операторов:

```
<тип> operator op (<тип> x, <тип> y)
```

Данные операторы применимы к аргументам следующих типов: **int**, **uint**, **long**, **ulong**, **bool** и перечислений.

Синтаксис побитового дополнения:

```
<тип> operator ~ (<тип> x)
```

Допустимые типы – те же самые за исключением **bool**.

Также побитовая конъюнкция и дизъюнкция применимы к обнуляемому типу **bool**?. При этом, если один из операндов имеет неопределенное значение (**null**), правила вычисления результата таковы:

```
false && null = null && false = false
true || null = null || true = true
```

Во всех остальных случаях результатом будет **null**.

Синтаксис операторов сдвига:

```
<тип> operator op (<тип> x, int y)
```

Допустимые типы аргумента – **int**, **uint**, **long** и **ulong**, а величина сдвига всегда имеет тип **int**. Сдвиг влево на n позиций эквивалентен умножению целого числа на 2^n, сдвиг вправо – целочисленному делению на 2^n. При сдвиге вправо чисел со знаком бит знака сохраняется. Во всех остальных случаях освободившиеся позиции заполняются нулями.

Пример:

```
bool? v13 = null;

Console.WriteLine("null & false = " + (v13 & false));
Console.WriteLine("true | null = " + (true | v13));
Console.WriteLine("null & false = " + (v13 & null));
// 0111 0111 0011 0101 1001 0100 0000 0000
v1 = 2000000000;
// 0011 1011 1001 1010 1100 1010 0000 0000
Console.WriteLine(v1 >> 1);
// 1110 1110 0110 1011 0010 1000 0000 0000
Console.WriteLine(v1 << 1);
// 1000 1000 1100 1010 0110 1100 0000 0000
v1 = -2000000000;
// 1100 0100 0110 0101 0011 0110 0000 0000
Console.WriteLine(v1 >> 1);
// 0001 0001 1001 0100 1101 1000 0000 0000
Console.WriteLine(v1 << 1);
```

Вывод на консоль:

```
null & false = False
true | null = True
null & false =
1000000000
-294967296
-1000000000
294967296
```

В третьей строке результат **null** выводится на консоль в виде пустой

строки.

3.3.3.5. Операторы сравнения

Большинство операторов возвращает числовые значения. Что касается операторов сравнения, они генерируют булевский результат. Операторы сравнения анализируют соотношение между операндами и возвращают значение **true**, если соотношение истинно, **false** – если ложно.

К операторам сравнения относятся операторы: меньше (<), меньше или равно (<=), больше (>), больше или равно (>=), равно (==) и не равно (!=). Синтаксис:

```
bool operator op (<тип> x, <тип> y)
```

Допустимые типы – **uint**, **int**, **long**, **ulong**, **float**, **double**, **decimal** и перечисления. Также операторы «==» и «!=» определены для типов **bool**, **object**, **string** и делегатов, а для обнуляемых типов они допускают сравнение с **null**.

На операциях со строками мы еще остановимся в п. 3.3.4, с делегатами – в § 4.8. Здесь лишь еще раз отметим, что при сравнении типов по значению сравнивается содержимое объектов, а при сравнении ссылочных типов – ссылки, хотя некоторые классы (типа System.String) эти операторы перегружают. Операторы «==» и «!=» используют метод Object.Equals, который для ссылочных типов эквивалентен методу Object.ReferenceEquals.

Пример:

```
int? v14 = null;
int val1 = 1;
int val2 = 1;
object obj1 = val1;
object obj2 = val2;

Console.WriteLine("v14 == null? " + (v14 == null));
Console.WriteLine("v13 == v11? " + (v13 == v11));
Console.WriteLine("v13 < v11? " + (v13 < v11)); // Ошибка
Console.WriteLine("val1 == val2? " + (val1 == val2));
Console.WriteLine("obj1 == obj2? " + (obj1 == obj2));
```

Вывод на консоль:

```
v14 == null? True
v13 == v11? False
val1 == val2? True
obj1 == obj2? False
```

3.3.3.6. Операторы присваивания

Выражения, которые могут находиться в левой части оператора присваивания, называются l-value, а в правой части – r-value. В качестве r-value может быть любая константа, переменная, число или выражение, результат которого совместим с l-value. Между тем, l-value должно быть переменной, свойством или индексатором определенного типа – должна существовать возможность вычислить ее физический адрес. Например, можно написать i = 1, но нельзя написать 1 = i или i + j = 1.

Операторы присваивания делятся на простой (=) и составные (+=, −=, *=, /=, %=, &=, |=, ^=, <<=, >>=).

Синтаксис простого оператора присваивания:

```
<тип> operator = (<тип> <l-value>, <тип> <r-value>)
```

Обратите внимание, что оператор присваивания возвращает результат (значение l-value). Его можно использовать в других выражениях, например, a = b = c, f(p = q), z = 4 + (x = y).

Оператор присваивания определен для всех объектов .NET. Как уже отмечалось, для типов по значению он копирует содержимое объектов, а для ссылочных типов – ссылки.

Синтаксис составных операторов присваивания:

```
<тип1> operator op= (<тип2> <rvalue>)
```

Аналогично, аргумент, использующийся в правой части данных операторов, должен быть l-value. Ограничения на типы данных – такие же, как для базовых операторов «op».

Хотя выражение x op= y эквивалентно выражению x = x op y, код MSIL в обоих случаях будет отличаться. Для составных операторов он будет более эффективным, поэтому используйте их везде, где это возможно.

Пример:

```
Console.WriteLine("v5 = 1 = " + (v5 = 1).GetType());
Console.WriteLine("v9 = v5 = " + (v9 = v5).GetType());
val1 = val2;
obj1 = obj2;
Console.WriteLine(val1 == val2 ? "val1 == val2" : "val1 != val2");
Console.WriteLine((object)val1 == (object)val2 ?
      "ref val1 == ref val2" : "ref val1 != ref val2");
Console.WriteLine(obj1.Equals(obj2) ? "obj1 == obj2" : "obj1 != obj2");
Console.WriteLine(Object.ReferenceEquals(obj1, obj2) ?
      "ref obj1 == ref obj2" : "ref obj1 != ref obj2");
```

Вывод на консоль:

```
v5 = 1 = System.Int64
v9 = v5 = System.Decimal
val1 == val2
ref val1 != ref val2
obj1 == obj2
ref obj1 == ref obj2
```

3.3.3.7. Операторы преобразования типа

В языке C# предусмотрены два оператора явного преобразования типа:

```
<тип1> operator <тип1> (<тип2> x)
<тип1> operator as (<тип2> x, <тип1>)
```

Если явное преобразование отсутствует, возникает ошибка компиляции. Отличия между ними следующие:

1) Оператор as осуществляет преобразование только к ссылочным типам данных;

2) Если в процессе преобразования возникают ошибки, оператор «()» вызывает исключительную ситуацию, а оператор as просто возвращает **null**.

Пример:

```
string v15;
object[] mas = new object[] { 1, "ABC", 2.5 };

Console.WriteLine(mas[0].GetType());
Console.WriteLine(mas[1].GetType());
Console.WriteLine(mas[2].GetType());
v1 = mas[0]; // Ошибка
v1 = (int)mas[0];
v1 = (int)mas[1]; // System.InvalidCastException
v1 = mas[2] as double; // Ошибка
v15 = mas[1] as string;
Console.WriteLine(v15);
Console.WriteLine(Object.ReferenceEquals(v15, mas[1]));
```

В примере создается массив объектов, первый из которых имеет тип **int**, второй – **string**, третий – **double**. Неявное преобразование из типа **object** в тип **int** не определено, поэтому первая операция присваивания вызывает ошибку компиляции. Вторая операция выполняется корректно, v1 и mas[0] имеют тип **int**. Третья вызывает исключительную ситуацию во время выполнения программы, т.к. невозможно преобразовать "ABC" к типу **int**. Четвертая операция вызывает ошибку компиляции – оператор as не выполняет преобразование к типам по значению. В пятой происходит корректное копирование ссылки на строку. Вывод на консоль:

```
System.Int32
System.String
System.Double
ABC
True
```

3.3.3.8. Операторы выбора

В языке C# есть два оператора, позволяющие осуществить выбор:

1) Условный оператор – единственный тернарный оператор (имеющий три аргумента). Синтаксис:

```
<тип> operator ?: (bool z, <тип1> x, <тип2> y)
```

Если значение «z» истинно, то результатом будет «x», иначе – «y». Тип результата – это наименьший общий тип для <тип1> и <тип2>, т.е. данные типы должны иметь неявное преобразование к типу результата. Если такое преобразование отсутствует, получаем ошибку компиляции. Обычно <тип> – это один из типов <тип1> и <тип2> с более широким диапазоном значений.

2) Оператор слияния с **null**. Синтаксис:

```
<тип1> operator ?? (<тип1> x, <тип2> y)
```

Здесь <тип1> должен быть ссылочным или обнуляемым, а <тип2> должен иметь неявное преобразование к <тип1>. Результатом операции будет «x», если x != **null**, иначе – «y».

Пример:

```
Console.WriteLine((v11 ? v1 : v5).GetType());
Console.WriteLine((v11 ? v5 : v6).GetType()); // Ошибка
v1 = v14 ?? -1;
Console.WriteLine(v1);
v15 = null;
Console.WriteLine(v15 ?? "строка пуста!");
v1 = v2 ?? 1; // Ошибка
```

Первая ошибка связана с тем, что не существует неявного преобразования между типами **long** и **ulong**. Вторая – v2 не является ссылочным или обнуляемым типом. Вывод на консоль:

```
System.Int64
-1
строка пуста!
```

3.3.3.9. Рефлективные операторы

Отражение (reflection) – это способность получать метаданные (т.е. данные о данных, или информацию о типе) в период исполнения (см. § 5.3).

Т.к. все объекты языка C# являются классами, то соответствующие метаданные называют метаклассами. Метаклассы представлены типом System.Type.

Синтаксис рефлективных операторов:

```
Type operator typeof (<тип>)
int operator sizeof (<тип>)
bool operator is (<тип1> x, <тип2>)
```

Оператор **typeof** возвращает метакласс типа. Также его можно получить, вызвав метод GetType() для экземпляра класса. Оператор **sizeof** возвращает количество байтов, занимаемых экземпляром указанного типа по значению на стеке. Оператор **is** проверяет, является ли «x» экземпляром типа <тип2> или типа, наследуемого от него.

Пример:

```
Console.WriteLine(v1.GetType());
Console.WriteLine(typeof(string));
Console.WriteLine(typeof(float[,]));
Console.WriteLine(typeof(float[,]).BaseType);
Console.WriteLine(typeof(void));

unsafe
{
      Console.WriteLine(sizeof(float));
      Console.WriteLine(sizeof(decimal));
      Console.WriteLine(sizeof(char));
      Console.WriteLine(sizeof(DateTime));
      Console.WriteLine(sizeof(string)); // Ошибка
}

Console.WriteLine((object)v1 is ValueType);
Console.WriteLine((object)v1 is String);
Console.WriteLine((object)v12 is Enum);
Console.WriteLine(mas[1] is String);
```

Вывод на консоль:

```
System.Int32
System.String
System.Single[,]
System.Array
System.Void
4
16
2
8
True
False
True
True
```

Класс Type имеет множество членов для получения полной информации о типе данных (подробности см. в библиотеке MSDN).

Оператор **sizeof** можно применять только в небезопасном коде. Для этого используется блок **unsafe** (см. § 5.5). Также необходимо разрешить использование небезопасного кода в проекте (пункт меню «Проект» → «Свойства…», вкладка «Построение», отметить опцию «Разрешить небезопасный код»). Для структур размером будет сумма размеров полей (с учетом выравнивания). Ошибка связана с попыткой определения размера типа по ссылке. Если структура будет содержать поля, имеющие ссылочный тип, это также вызовет ошибку компиляции.

3.3.3.10. Операторы управления переполнением

Данные операторы управляют контекстом проверки переполнения для арифметических операций и преобразований с целыми типами данных. Синтаксис:

```
<тип> operator checked (<тип> <выражение>)
<тип> operator unchecked (<тип> <выражение>)
```

Оператор **checked** включает проверку переполнения при вычислении целочисленных выражений, а оператор **unchecked** отключает ее. По умолчанию проверка выключена. Также данные операторы можно использовать как операторы языка, за которыми следует блок программы:

```
checked <блок>
unchecked <блок>
```

В этом случае их действие распространяется на весь блок. Пример использования рассмотрен в п. 3.1.2.8.

3.3.3.11. Операторы для работы с указателями

При работе с указателями применяются многие из операций, знакомых ещё из языка C++:

- «*» – разыменование указателя;
- «->» – доступ к члену структуры, представленной указателем;
- «[]» – индексирование указателя;
- «&» – получение адреса переменной.

Как и в языке C++, к указателям можно применять некоторые арифметические операции (+, –, ++, ––) и операции сравнения. Работа с указателями разрешена только в небезопасном коде. Подробнее мы ее рассмотрим в § 5.5.

3.3.3.12. Лямбда-оператор

Маркер «=>» называется лямбда-оператором. Он используется в лямбда-выражениях для отделения входных переменных с левой стороны от тела лямбда-выражения с правой стороны. Лямбда-выражения – это встроенные выражения, аналогичные анонимным методам (см. § 4.8), но более гибкие. Они широко используются в запросах LINQ, выраженных с использованием синтаксиса методов. Материал по лямбда-выражениям в данное учебное пособие не входит.

3.3.4. Операции со строками

Как уже отмечалось в п. 3.1.3.1, в классе System.String определены четыре оператора для работы со строками – «==», «!=», «+» и «[]» (последний из них, строго говоря, является в терминах языка C# не оператором, а индексатором). Как и все объекты, строки также допускают присваивание («=»). Как будет показано в § 4.7, если к экземплярам какого-либо класса применим оператор «+», то к ним также применим оператор «+=». Это относится и к строкам. Строки не допускают модификации содержимого, поэтому операция «x += y» фактически не изменяет строку «x», а создает новую строку «x + y» и помещает ссылку на нее в «x» (см. пример ниже).

Как видно, из операций отношения к строкам применимы только операции «равно» и «не равно». Их действие основано на вызове метода Equals. Как реализовать другие операции отношения? Для этого используются различные варианты методов Compare/CompareOrdianl/CompareTo. Некоторые из них статические (например, String.Compare(x, y)), другие – нет (например, x.CompareTo(y)). Результат, возвращаемый этими методами, зависит от соотношений аргументов:

• отрицательное число, если x < y;

• ноль, если x = y;

• положительное число, если x > y.

Поэтому, например, для выполнения операции отношения «x ≤ y» следует писать «String.Compare(x, y) ≤ 0».

Пример:

```
string copy = "Это исходная строка";
string x = copy;
```

```
Console.WriteLine(Object.ReferenceEquals(copy, x));
x += "!!!";
Console.WriteLine(Object.ReferenceEquals(copy, x));
Console.WriteLine(copy);
Console.WriteLine(x);
Console.WriteLine(copy == "Это исходная строка");
Console.WriteLine(copy.Equals("Это исходная строка"));
Console.WriteLine(Object.Equals(copy, "Это исходная строка"));
Console.WriteLine(String.Equals(copy, "Это исходная строка"));
Console.WriteLine(copy == x);
Console.WriteLine(String.Compare(x, copy));
Console.WriteLine(copy.CompareTo(x));
```

Вывод на консоль:

```
True
False
Это исходная строка
Это исходная строка!!!
True
True
True
True
False
1
-1
```

 Пример: Samples\3.3\3_3_4_string.

3.3.5. Операции с перечислениями

С переменными перечисляемого типа можно выполнять арифметические операции (+, -, ++, --), логические поразрядные операции (^, &, |, ~), сравнивать их с помощью операций отношения (<, <=, >, >=, ==, !=) и получать размер в байтах (sizeof). При использовании переменных перечисляемого типа в целочисленных выражениях и операциях присваивания требуется явное преобразование типа.

Пример:

```
[FlagsAttribute] enum BitFlags : byte
{
      bit1 = 0x01, bit2 = 0x02, bit3 = 0x04, bit4 = 0x08,
      bit5 = 0x10, bit6 = 0x20, bit7 = 0x40, bit8 = 0x80
}

static int Main()
{
      BitFlags bits = (BitFlags)29;
      Array values = Enum.GetValues(bits.GetType());
      DayOfWeek day = DayOfWeek.Friday;
      DayOfWeek next = day + 1;
      CultureInfo ru = new CultureInfo("ru-RU");
      string[] days = ru.DateTimeFormat.DayNames;
```

```
            Console.Write("0x{0:X} = ", (int)bits);

            for (int i = values.Length - 1; i >= 0; i--)
            {
                  Console.Write((bits & (BitFlags)values.GetValue(i)) != 0 ?
"1" : "0");
            }

            Console.WriteLine("Если сегодня {0}, то завтра будет {1}",
                  days[(int)day], days[(int)next]);
            next -= 3;
            Console.WriteLine("А {0} {1}, чем {2}", days[(int)day],
                  day < next ? "раньше" : "позже", days[(int)next]);
            return 0;
      }
```

Вывод на консоль:

```
0x1D = 00011101
Если сегодня пятница, то завтра будет суббота
А пятница позже, чем среда
```

 Пример: Samples\3.3\3_3_5_enum.

3.3.6. Операции с типом DateTime

Согласно табл. А.1 в приложении А, для структуры DateTime определены операции сложения, вычитания и все операции отношения. Оператор «+» применим для прибавления к дате и времени интервала (выраженного экземпляром структуры TimeSpan). Оператор «-» позволяет найти интервал между двумя датами, или отнять от даты интервал. Операторы «+=» и «-=» работают только с экземплярами даты и времени в левой части и интервалами в правой.

Пример:

```
      DateTime date1 = new DateTime(2010, 10, 10);
      DateTime date2 = new DateTime(2007, 7, 7);
      TimeSpan span = date1 - date2;
      DateTime date3 = date2 - span;

      Console.WriteLine("Между {0:d} и {1:d} - {2} дней", date1, date2,
span.TotalDays);
      Console.WriteLine("За {0} дней до {1:d} было {2:d}", span.TotalDays,
date2, date3);
      date3 = date1;
      date3 += span;
      Console.WriteLine("Через {0} дней после {1:d} будет {2:d}",
span.TotalDays, date1, date3);
      date3 = DateTime.Today;

      if (date3 == date2) Console.WriteLine("Сегодня 7 июля 2007 года!");
      else if (date3 < date2) Console.WriteLine("7 июля 2007 года еще не
```

```
наступило!");
      else Console.WriteLine("7 июля 2007 года уже прошло!");
```

Вывод на консоль:

```
Между 10.10.2010 и 07.07.2007 - 1191 дней
За 1191 дней до 07.07.2007 было 02.04.2004
Через 1191 дней после 10.10.2010 будет 13.01.2014
7 июля 2007 года уже прошло!
```

 Пример: Samples\3.3\3_3_6_datetime.

3.3.7. Математические вычисления

Вычисление математических выражений является важным элементом языка. Основной набор перечисленных здесь функций имеется и в математических библиотеках языка C++, хотя от поддержки чисел типа **long double** (используемые в Borland C++ 10-байтовые числа с плавающей точкой и диапазоном $\pm 3.6 \cdot 10^{-4951}$… $\pm 1.1 \cdot 10^{4932}$, поддерживаемые на аппаратном уровне математическими сопроцессорами) в языке C# разработчики отказались.

 Пример: Samples\3.3\3_3_7_math.

3.3.7.1. Математические функции и константы

Все математические функции и константы в языке C# реализованы в виде членов класса System.Math. Список его открытых членов приведен в табл. 3.19. Все они являются статическими.

Табл. 3.19 – Члены класса System.Math

Член	Описание
Методы	
static <тип> Abs(<тип> value)	Возвращает абсолютное значение числа(1)
static double Acos(**double** d)	Арккосинус числа (arccos *d*)
static double Asin(**double** d)	Арксинус числа (arcsin *d*)
static double Atan(**double** d)	Арктангенс числа (arctg *d*)
static double Atan2(**double** y, **double** x)	Арктангенс отношения y/x (корректно работает для x = 0)
static long BigMul(**int** a, **int** b)	Умножает два 32-битовых числа
static <тип> Ceiling(<тип> d)	Округление числа(2, 4)

static double Cos(**double** d)	Косинус угла (cos *d*)
static double Cosh(**double** d)	Гиперболический косинус угла (cosh *d*)
static <тип> DivRem(<тип> a, <тип> b, **out** <тип> result)	Возвращает частное от деления и остаток в выходном параметре для типов **int** и **long**
static double Exp(**double** d)	Возводит число *e* в указанную степень (e^d)
static <тип> Floor(<тип> d)	Округление числа[2, 4]
static double IEEERemainder (**double** x, **double** y)	Возвращает остаток от деления x на y
static double Log(**double** d)	Натуральный логарифм (по основанию *e*) (ln *d*)
static double Log(**double** a, **double** base)	Логарифм по указанному основанию ($\log_{base} d$)
static double Log10(**double** d)	Десятичный логарифм (lg *d*)
static <тип> Max(<тип> val1, <тип> val2)	Возвращает наибольшее из двух чисел[3]
static <тип> Min(<тип> val1, <тип> val2)	Возвращает наименьшее из двух чисел[3]
static double Pow(**double** x, **double** y)	Возведение в степень (x^y)
static <тип> Round(<тип> d, ...)	Ряд методов для округления чисел[2, 4]
static int Sign(<тип> value)	Знак числа[1]. Возвращает 1 для положительных чисел, 0 для нуля, –1 для отрицательных чисел
static double Sin(**double** d)	Синус угла (sin *d*)
static double Sinh(**double** d)	Гиперболический синус угла (sinh *d*)
static double Sqrt(**double** d)	Квадратный корень
static double Tan(**double** d)	Тангенс угла (tg *d*)
static double Tanh(**double** d)	Гиперболический тангенс угла (tgh *d*)
static decimal Truncate(<тип> d)	Округление числа[2, 4]
Поля	
const double E	Основание натурального логарифма

	(число *e*)
const double PI	Отношение длины окружности к ее диаметру (число π)

Примечания:

(1) Для типов **sbyte**, **short**, **int**, **long**, **float**, **double** и **decimal**;

(2) Для типов **double** и **decimal**;

(3) Для всех целых типов и типов с плавающей точкой;

(4) Различия в способах округления чисел представлены в табл. 3.20.

Табл. 3.20 – Сравнение методов округления чисел

	–7.9	–7.1	7.1	7.9
Ceiling	–7	–7	8	8
Floor	–8	–8	7	7
Round	–8	–7	7	8
Truncate	–7	–7	7	7

Пример:

```
      double angle = 30;
      double rad = angle * Math.PI / 180;

      Console.WriteLine("Угол {0}° в радианах составит {1} = pi/{2}", angle,
rad, 180/angle);
      Console.WriteLine("cos {0}° = {1}", angle, Math.Cos(rad));
      Console.WriteLine("sin {0}° = {1}", angle, Math.Sin(rad));
```

Вывод на консоль:

```
      Угол 30° в радианах составит 0,523598775598299 = pi/6
      cos 30° = 0,866025403784439
      sin 30° = 0,5
```

3.3.7.2. Генерация случайных чисел

Часто в математических вычислениях требуются случайные числа (для реализации различных стохастических алгоритмов, вариантов методов Монте-Карло, получения распределения Гаусса и т.п.). Для получения последовательностей псевдослучайных чисел в языке C# используются экземпляры класса System.Random.

При генерации таких последовательностей вводится понятие *номера последовательности*. Последовательности являются не полностью случайными, а псевдослучайными, т.е. их генерация зависит от некоторого начального значения.

Список членов класса Random приведен в табл. 3.21.

Табл. 3.21 – Члены класса System.Random

Члены	Описание
Конструкторы	
`Random()`	Инициализирует новый экземпляр класса, номер последовательности зависит от текущего времени (аналог в языке C++ – randomize() или srand(gettime()))
`Random(int Seed)`	Инициализирует новый экземпляр класса с указанным номером последовательности (аналог srand(seed))
Методы	
`int Next()`	Генерирует следующее число последовательности
`int Next(int maxValue)`	Для maxValue > 1 генерирует следующее число последовательности в диапазоне [0, maxValue – 1]
`int Next(int minValue, int maxValue)`	Генерирует следующее число последовательности, лежащее в диапазоне [minValue, maxValue – 1]
`void NextBytes (byte[] buffer)`	Заполняет элементы указанного массива байтов случайными числами в диапазоне [0, 255]
`double NextDouble()`	Возвращает случайное число с плавающей точкой в диапазоне $0 \le x < 1$

Пример:

```
Random rnd = new Random();

Console.WriteLine("10 случайных целых чисел в диапазоне [-100,100]:");

for (int i = 0; i < 10; i++)
{
      Console.Write("{0} ", rnd.Next(-100, 101));
}

Console.WriteLine();
```

Вывод на консоль:

```
10 случайных целых чисел в диапазоне [-100,100]:
17 48 69 37 -14 84 -42 55 -64 88
```

§ 3.4. Операторы языка

Язык C# содержит множество операторов. Большинство из них будут знакомы разработчикам, имеющим опыт программирования на языках C и C++.

3.4.1. Основные понятия

 Пример: Samples\3.4\3_4_1_csoper.

3.4.1.1. Операторы и блоки операторов

Операторы языка C# делятся на оператор объявления переменной, помеченный оператор и группу *внедряемых операторов*:

```
<оператор> :: <оператор объявления>
<оператор> :: <помеченный оператор>
<оператор> :: <внедряемый оператор>
```

Остановимся на них подробнее.

1) Оператор объявления переменной (§ 3.1):

```
<оператор объявления> :: <тип данных> <список переменных с
инициализацией>;
```

2) Помеченный оператор (п. 3.4.4.3):

```
<помеченный оператор> :: <метка>: <оператор>
```

Все перечисленные ниже операторы являются внедряемыми.

3) Пустой оператор:

```
<пустой оператор> :: ;
```

4) Блок операторов:

```
<блок> :: "{" <список операторов> "}"
<блок> :: "{" "}"
```

Как видно, блок может быть пустым. Список операторов всегда заключается в операторные скобки «{}», за единственным исключением – в ветках оператора выбора (см. п. 3.4.2.2).

5) Оператор выражения (§ 3.3):

```
<выражение> :: <вычисляемое выражение>
```

6) Операторы управления ходом выполнения программы. Разбиты на три категории: операторы ветвления (п. 3.4.2), операторы цикла (п. 3.4.3) и

операторы перехода (п. 3.4.4). Во всех выполняется проверка вычисленного булевского значения, и на основе этой проверки изменяется ход выполнения программы. Также к данным операторам можно отнести операторы для работы с исключительными ситуациями (п. 3.4.5);

7) Операторы проверки переполнения. Как мы уже отмечали (см. п. 3.3.3.10), операторы **checked** и **unchecked** могут выступать как в роли операторов вычисления выражений, так и в роли операторов языка:

```
<оператор проверки переполнения> :: checked <блок>
<оператор проверки переполнения> :: unchecked <блок>
```

8) Оператор блокировки (п. 5.2.3):

```
<оператор блокировки> :: lock (<выражение>) <внедряемый оператор>
```

9) Оператор управления ресурсами (см. п. 3.5.3):

```
<оператор выделения ресурсов> :: using (<оператор объявления>)
<внедряемый оператор>
```

Требуется, чтобы переменные в объявлении обязательно были инициализированы, а тип данных реализовал интерфейс System.IDisposable.

Внедряемые операторы – это операторы, которые могут входить в состав других операторов языка. Здесь не рассматриваются еще два внедряемых оператора – **unsafe** и **fixed**. О них речь пойдет в § 5.5. Некоторые конструкции языка C# требуют использования внедряемых операторов, например, условный оператор **if**. Так, запись

```
bool x = true;
if (x) int i = 1; // Ошибка
```

будет вызывать ошибку компилятора, т.к. объявление переменной не является внедряемым оператором. С другой стороны, если объявление переменной заключить в блок или использовать только оператор выражения, ошибки не будет:

```
int j;

if (x)
{
      int i = 1; // OK
}

if (x) j = 1; // OK
```

Другие операторы, например, **checked** и **unchecked**, требуют обязательного использования блока операторов:

```
uint k = 0;
checked k--; // Ошибка
unchecked { k--; } // ОК
```

3.4.1.2. Конечные точки и достижимость

У каждого оператора языка имеется *конечная точка* – это позиция, непосредственно следующая за оператором. Когда управление достигает конечной точки оператора в блоке, оно передается следующему оператору этого блока.

Если существует возможность передачи управления оператору языка в ходе выполнения, говорят, что он является *достижимым*. И наоборот, если возможность выполнения оператора исключена, он называется *недостижимым*.

Пример:

```
public static int Test()
{
      const bool f = false;
      if (f) Console.WriteLine("b = true!"); // Предупреждение
      goto x;
      Console.WriteLine("Конец..."); // Предупреждение
x:    return 0;
      Console.WriteLine("...функции Test!"); // Предупреждение
}
```

Здесь все три метода WriteLine являются недостижимыми. Первый – потому что условие всегда ложно (f – константа), второй – потому что перед ним стоит оператор безусловного перехода, третий – потому что стоит после оператора возврата из функции.

Для недостижимых операторов компилятор C# выдает предупреждение «Обнаружен недостижимый код». В двух ситуациях достижимость конечной точки означает ошибку компиляции:

• Достижимость конечной точки списка операторов ветки **case** в операторе **switch**. Обычно такая ошибка свидетельствует об отсутствии оператора **break** (см. п. 3.4.2.2).

• Достижимость конечной точки метода, возвращающего значение. Обычно это свидетельствует об отсутствии оператора **return** (см. п. 3.4.4.4).

3.4.2. Операторы ветвления

Позволяют определить, когда и какой код выполнять. В языке C# предусмотрены два оператора выбора: **switch**, управляющий ветвлением про-

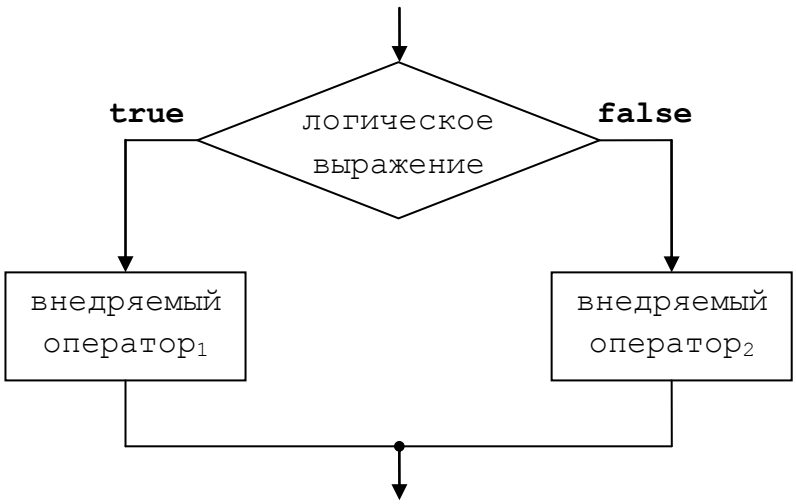
граммы на основе некоторого значения, и **if**, который выполняет код в зависимости от логического условия.

Пример: Samples\3.4\3_4_2_cond.

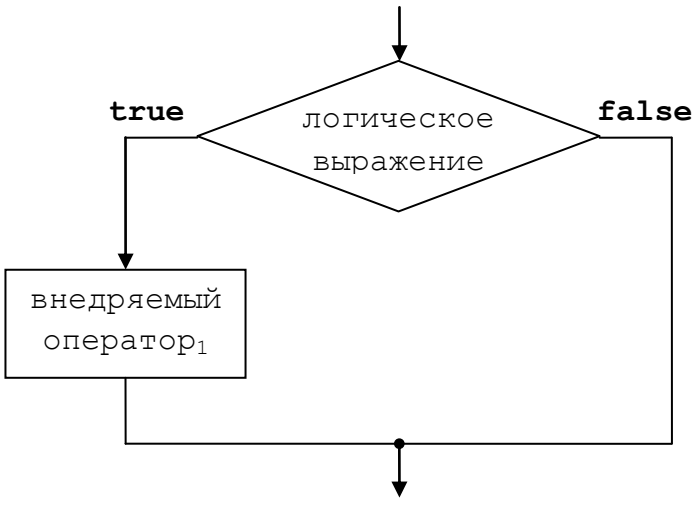
3.4.2.1. Условный оператор if

Выполняет блок операторов, если вычисленное им выражение имеет результат **true**. Синтаксис:

```
<условный оператор> ::
if (<логическое выражение>) <внедряемый оператор1>
[else <внедряемый оператор2>]
```



а) с веткой **else**



б) без ветки **else**

Рис. 3.8 – Схема работы условного оператора

Как и в языке C++, ветка **else** является необязательной (рис. 3.8), а в качестве внедряемого оператора может, в свою очередь, использоваться другой условный оператор.

Если используются вложенные условные операторы, то оператор **else** будет относиться к ближайшему оператору **if**. Например, следующий код

```
if (логическое выражение1)
if (логическое выражение2) блок1
else блок2
else блок3
```

соответствует

```
if (логическое выражение1)
{
      if (логическое выражение2) блок1
      else блок2
}
else блок3
```

Возможен вариант с несколькими конструкциями **else**:

```
if (логическое выражение1) блок1
else if (логическое выражение2) блок2
else if (логическое выражение3) блок3
else ...
else if (логическое выражениеn) блокn
else блокn+1
```

Это соответствует следующему коду:

```
if (логическое выражение1) блок1
      else
      {
            if (логическое выражение2) блок2
            else
            {
                  if (логическое выражение3) блок3
                  else
                  {
                        ...
                        else
                        {
                              if (логическое выражениеn) блокn
                              else блокn+1
                        }
                  }
            }
      }
```

Пример:

```
int b;

Console.Write("Введите целое число: ");

if (int.TryParse(Console.ReadLine(), out b))
```

```
{
      if (b < 0) Console.WriteLine("Вы ввели отрицательное число");
      else if (b > 0) Console.WriteLine("Вы ввели положительное число");
      else Console.WriteLine("Вы ввели ноль");
}
else
{
      Console.WriteLine("Введено некорректное число");
}
```

3.4.2.2. Оператор выбора switch

Оператор **switch** выполняет список операторов, метка которого соответствует значению выражения выбора:

```
<оператор выбора> :: switch (<выражение выбора>)
"{"
      <метка1,1> [<метка1,2> ...] <список операторов1>
      <метка2,1> [<метка2,2> ...] <список операторов2>
      ...
      <меткаN,1> [<меткаN,2> ...] <список операторовN>
"}"
<оператор выбора> :: switch (<выражение выбора>) "{" "}"

<метка> :: case <константное выражение>:
<метка> :: default:
```

Схема его работы приведена на рис. 3.9. Замечания:

1. Пустой блок оператора **switch** допустим, хотя компилятор выдает для него предупреждение «Пустой блок switch».

2. Выражение выбора должно иметь тип **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **string** или перечисления, либо же должно иметь неявное преобразование в один из этих типов.

3. Константное выражение для ветки **case** должно иметь тип выражения выбора, либо неявное преобразование к этому типу.

4. Не допускается использование двух одинаковых меток. В противном случае компилятор генерирует ошибку «Метка уже встречалась в этом операторе выбора».

5. В списке операторов каждой ветки **case** конечная точка не должна быть достижима, иначе компилятор генерирует ошибку «Управление не может передаваться вниз от одной метки case к другой». Поэтому последним в списке должен быть оператор, не допускающий достижимость конечной точки, например:

• Оператор **break**, завершающий выполнение оператора **switch**;

• Оператор **goto** <метка>, осуществляющий переход на другую ветку:

```
goto case <константное выражение>;
goto default;
```

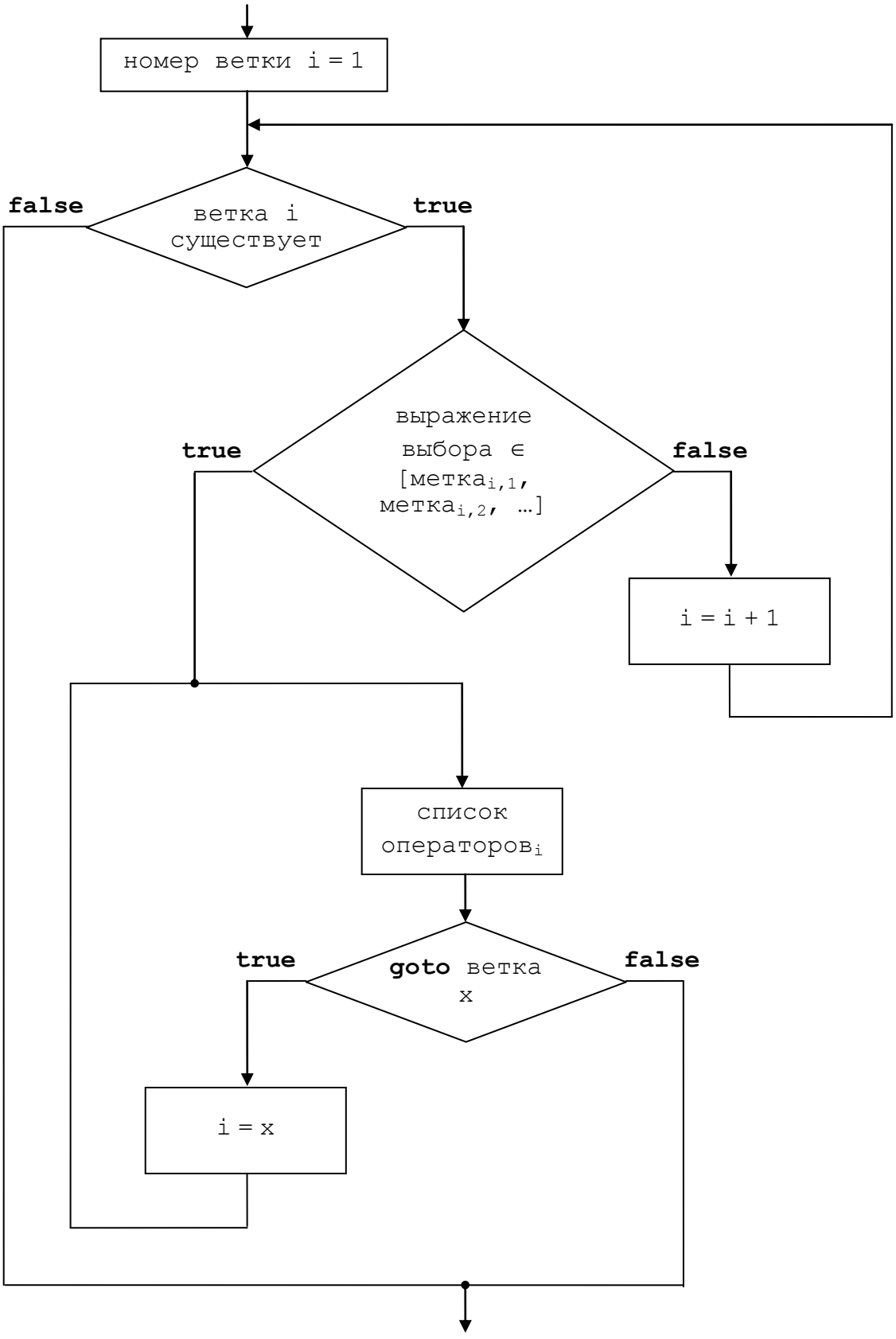


Рис. 3.9 – Схема работы оператора выбора

• Оператор **goto** <идентификатор>, осуществляющий безусловный переход (см. п. 3.4.4.3);

• Оператор **return**, завершающий выполнение текущего метода (см. п. 3.4.4.4);

• Оператор **throw**, передающий управление блоку обработки исключительной ситуации (см. п. 3.4.5);

• Любая другая конструкция, исключающая передачу управления следующему оператору, например, бесконечный цикл или оператор **continue**, если оператор **switch** находится внутри цикла.

Пример:

```
static int Main()
{
      DayOfWeek d = DayOfWeek.Friday;

      switch (true) { } // Предупреждение
      switch (1.0) { } // Ошибка

      switch (d)
      {
            default:
                  Console.WriteLine("Такого дня недели нет");
                  goto x;

            case 1: // Ошибка
            case (DayOfWeek)1: // ОК
                  Console.WriteLine("Понедельник");
                  break;

            case DayOfWeek.Monday: // Ошибка
                  Console.WriteLine("Понедельник");
                  break;

            default: // Ошибка
                  Console.WriteLine();
                  break;

            case DayOfWeek.Thursday: // Ошибка
                  Console.Write("Четверг");

            case DayOfWeek.Friday:
                  Console.Write("Пятница. А завтра будет... ");
                  goto case DayOfWeek.Friday + 2; // Ошибка
                  goto case DayOfWeek.Friday + 1;

            case DayOfWeek.Saturday:
            case DayOfWeek.Sunday:
                  Console.WriteLine("Выходной день!");
                  break;

            case (DayOfWeek)8:
                  goto default;
```

```
                case DayOfWeek.Tuesday:
                      while (true) // Здесь программа зациклится
                      {
                            Console.WriteLine("Вторник");
                      }

                case DayOfWeek.Wednesday:
                      Console.WriteLine("Среда");
                      return 0;

                case DayOfWeek.Thursday:
                      // Генерация исключительной ситуации
                      throw new Exception("Четверг");
          }

    x:    return 0;
    }
```

3.4.3. Операторы цикла

Управляемые итерации, или циклы, в языке C# выполняют операторы **while**, **do/while**, **for** и **foreach**. В каждом случае исполняется внедряемый оператор, пока условие цикла является истинным. Однако, выполнение любого цикла может также быть прервано одним из операторов перехода (см. п. 3.4.4).

Пример: Samples\3.4\3_4_3_iter.

3.4.3.1. Оператор while

Синтаксис оператора **while**:

```
<оператор while> :: while (<логическое выражение>) <внедряемый оператор>
```

Схема работы оператора **while** приведена на рис. 3.10.

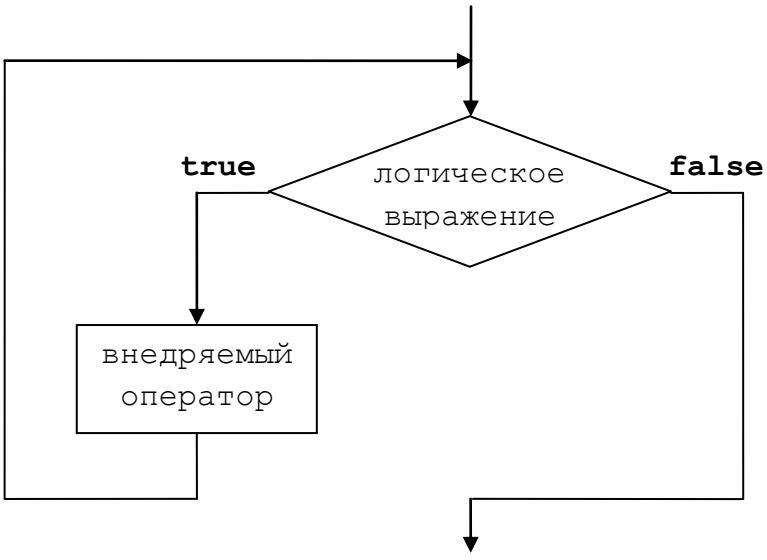


Рис. 3.10 – Схема работы оператора **while**

Пример:

```
int[] a = { 1, 15, 4, -2, 8, -6, -10 };
int i = 0;
int? neg = null;

while (i < a.Length)
{
      if (neg == null && a[i] < 0)
      {
            neg = a[i];
      }
      i++;
}

if (neg == null) Console.WriteLine("В массиве нет отрицательных чисел");
else Console.WriteLine("Первое отрицательное число {0}", neg);
```

3.4.3.2. Оператор do-while

Иногда, когда требуется, чтобы тело цикла обязательно было выполнено хотя бы один раз, использование оператора **while** становится неудобным. В этом случае целесообразнее применять оператор **do-while**:

```
      <оператор do-while> :: do <внедряемый оператор> while (<логическое
выражение>);
```

Схема его работы изображена на рис. 3.11.

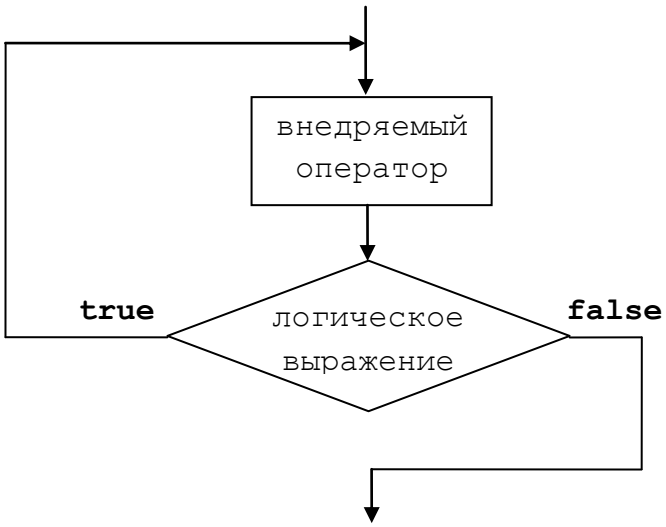


Рис. 3.11 – Схема работы оператора **do-while**

Пример:

```
Console.Write("Введите отрицательное число: ");
neg = null;
```

```
do
{
      if (int.TryParse(Console.ReadLine(), out i) && i < 0) neg = i;
      else Console.Write("Повторите ввод: ");
} while (neg == null);
```

3.4.3.3. Оператор for

Самый распространенный итерационный оператор. Синтаксис:

```
<оператор for> :: for ([<инициализатор>]; [<логическое выражение>];
[<итератор>]) <внедряемый оператор>

<инициализатор> :: <оператор объявления>
<инициализатор> :: <выражение1> [, <выражение2>] [, ...]

<итератор> :: <выражение1> [, <выражение2>] [, ...]
```

Все три части цикла (инициализатор, логическое выражение, итератор) являются необязательными. При отсутствии логического выражения условие цикла всегда считается истинным.

Схема работы цикла **for** изображена на рис. 3.12.

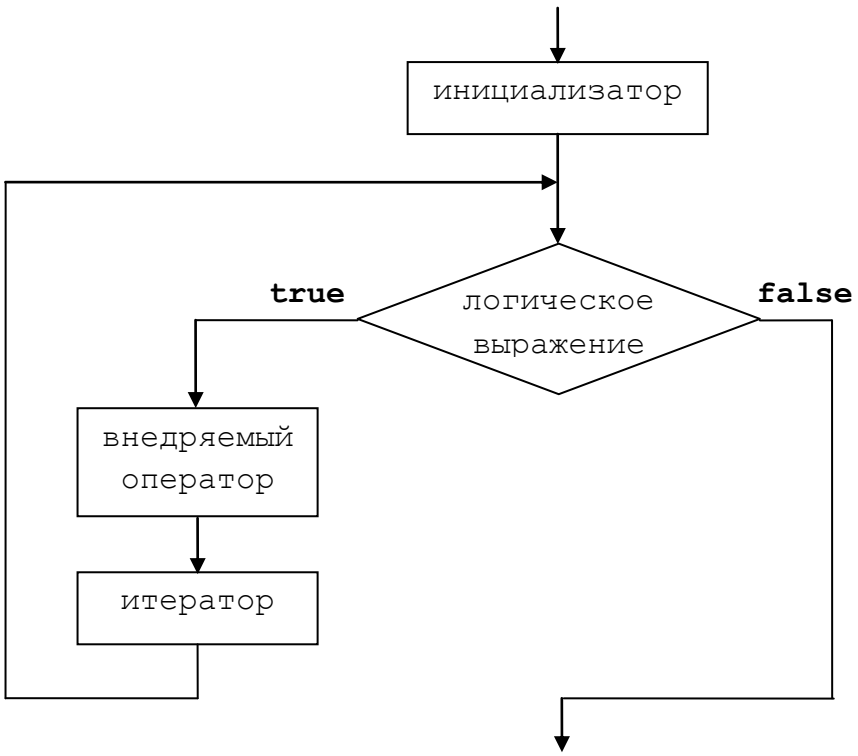


Рис. 3.12 – Схема работы оператора **for**

Пример:

```
int[,] mas = { { 15, 22, -3, -1 }, { -4, 0, 8, 1 }, { 2, -11, 3, 14 } };
int[] vec = new int[mas.GetLength(0) * mas.GetLength(1)];

for (int odd = 0, k = 0; odd <= 1; odd++)
```

```
    {
        for (i = 0; i < mas.GetLength(0); i++)
        {
            for (int j = 0; j < mas.GetLength(1); j++)
            {
                if (Math.Abs(mas[i, j] % 2) == odd)
                    vec[k++] = mas[i, j];
            }
        }
    }

    Console.Write("Четные и нечетные элементы матрицы: ");
    for (i = 0; i < vec.Length; i++) Console.Write("{0} ", vec[i]);
    Console.WriteLine();
```

3.4.3.4. Оператор foreach-in

В языке C# есть специальный оператор для перечисления элементов коллекций – оператор **foreach-in**.

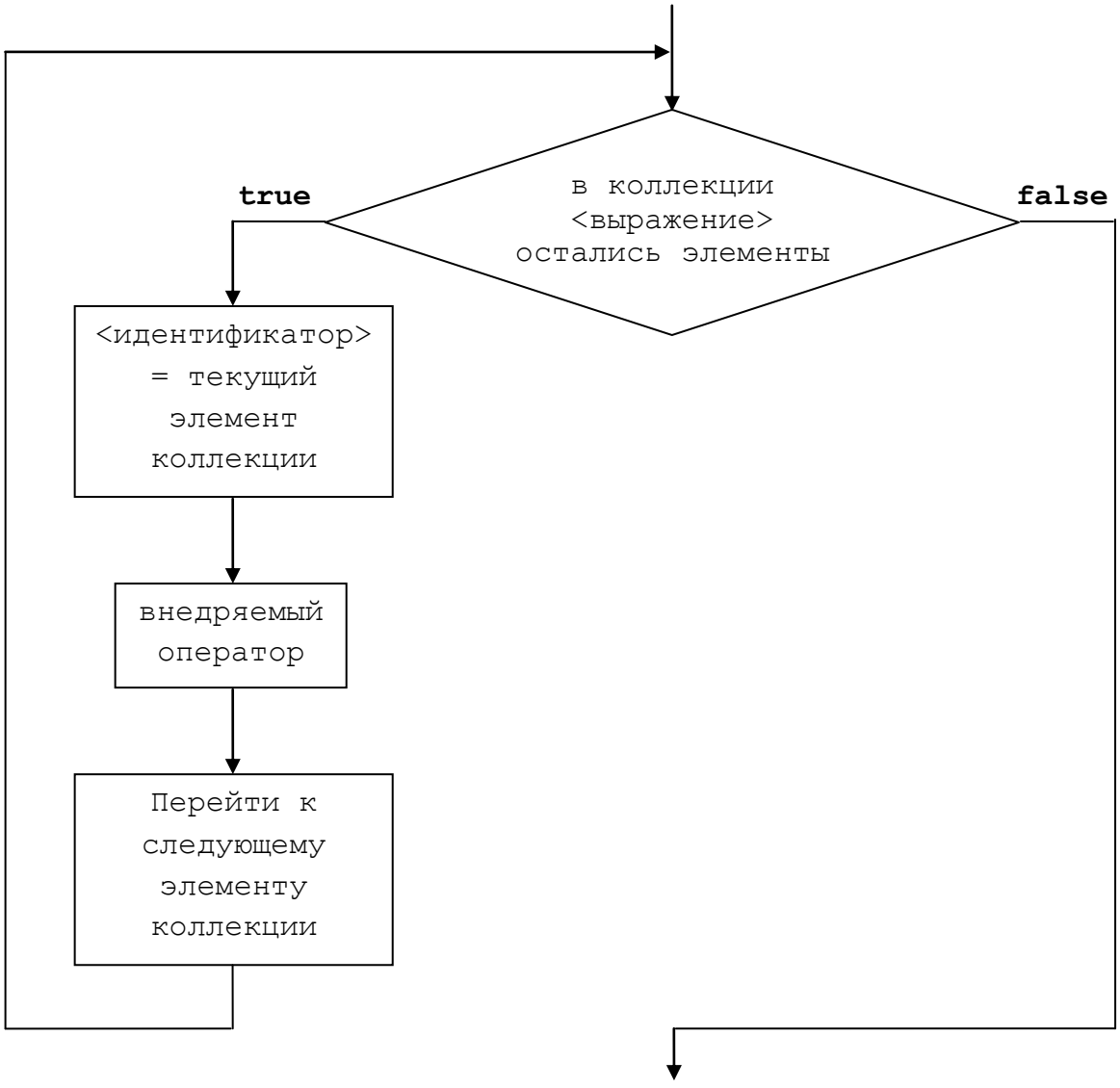


Рис. 3.13 – Схема работы оператора **foreach-in**

Синтаксис:

```
    <оператор foreach-in> :: foreach (<тип> <идентификатор> in <выражение>)
<внедряемый оператор>
```

Замечания:

1. Тип выражения должен являться коллекцией, а тип идентификатора – типом элементов этой коллекции. Обычно коллекция – это класс или структура, реализующая интерфейс IEnumerable. Как создавать свои собственные коллекции, мы разберемся в § 4.9. В частности, рассмотренный нами ранее класс System.Array реализует данный интерфейс, поэтому цикл **foreach** можно применять для обработки любых массивов.

2. В качестве типа идентификатора можно использовать **var**, тогда тип будет определяться автоматически.

Схема работы цикла **foreach-in** изображена на рис. 3.13. Пример:

```
    Console.Write("Все элементы матрицы: ");
    foreach (int elem in mas) Console.Write("{0} ", elem);
    Console.WriteLine();
```

3.4.4. Операторы перехода

Мы можем управлять ходом исполнения программы с помощью одного из операторов перехода: **break**, **continue**, **goto**, **return**, **throw** и **yield**. Все эти операторы осуществляют безусловную передачу управления.

Замечание. Если оператор безусловного перехода **break**, **continue**, **goto** или **return** находится внутри блока **try** или **catch** (см. п. 3.4.5), и пытается передать управление в точку вне этого блока, то сначала выполняется код блока **finally** (если он есть), а уже затем осуществляется передача управления (см. пример ниже). Из самого блока **finally** передача этими операторами управления запрещена.

 Пример: Samples\3.4\3_4_4_jump.

3.4.4.1. Оператор break

Прерывает текущий цикл или оператор выбора. После этого управление передается на строку кода, следующую за оператором цикла или оператора выбора. Синтаксис:

```
    <оператор break> :: break;
```

Пример:

```
int[] a = { 1, 15, 4, -2, 8, -6, -10 };
int i = 0;
int? neg = null;

Console.WriteLine("Начинаем обработку массива...");

while (i < a.Length)
{
      try
      {
            if (neg == null && a[i] < 0)
            {
                  neg = a[i];
                  Console.WriteLine("Отрицательное число найдено.
Прерываем цикл...");
                  break;
            }
      }
      finally
      {
            Console.WriteLine("{0}-я итерация цикла закончена...", i);
            i++;
      }
}

if (neg == null) Console.WriteLine("В массиве нет отрицательных чисел");
else Console.WriteLine("Найдено число {0}", neg);
```

3.4.4.2. Оператор continue

Как и **break**, оператор **continue** позволяет изменять выполнение цикла. Но **continue** не завершает оператор текущего цикла, а прерывает текущую итерацию и передает управление на начало цикла для следующей итерации. Синтаксис:

```
<оператор continue> :: continue;
```

Пример:

```
Console.Write("Введите отрицательное число: ");
neg = null;

do
{
      if (!int.TryParse(Console.ReadLine(), out i) || i >= 0)
      {
            Console.Write("Повторите ввод: ");
            continue;
      }
      neg = i;
} while (neg == null);
```

3.4.4.3. Оператор goto

Оператор **goto** передает управление оператору, обозначенному меткой. Синтаксис:

```
<оператор goto> :: goto <идентификатор>;
<оператор goto> :: goto case <константное выражение>;
<оператор goto> :: goto default;
```

Два последних варианта синтаксиса применяются в операторе выбора (см. п. 3.4.2.2).

Первый вариант может применяться для перехода на любые помеченные операторы. Однако, если оператор **goto** вызывается вне области видимости метки, при компиляции возникнет ошибка. Таким образом, он не может применяться для перехода в другой метод или внутрь вложенного цикла, хотя выйти из вложенного цикла можно. В целом, использование этой формы оператора **goto** не рекомендуется. Он нарушает структурность программы и ухудшает ее читабельность. В языке C# достаточно других операторов, позволяющих обойтись без оператора безусловного перехода.

3.4.4.4. Оператор return

Оператор **return** определяет значение, возвращаемое исполняемым в данный момент методом, и приводит к немедленному возврату к вызывающему оператору. Синтаксис:

```
<оператор return> :: return [<выражение>];
```

Замечания:

1. Если метод ничего не возвращает (имеет тип возвращаемого значения **void**), то оператор **return** используется без выражения;

2. Если метод имеет тип возвращаемого значения, отличный от **void**, и не все ветви кода метода возвращают значение оператором **return**, компилятор генерирует ошибку. Исключение – использование оператора **yield** в итераторах (см. п. 3.4.4.6).

3. Тип выражения должен совпадать с типом возвращаемого методом значения, или иметь неявное преобразование к нему.

4. Как мы уже отмечали, если оператор **return** находится внутри блока обработки исключительной ситуации (**try** или **catch**), то сначала выполняется код блока **finally** (если он описан), а уже затем осуществляется возврат.

Пример:

```
	static int Main()
	{
		try
		{
			Console.WriteLine("Завершаем выполнение программы...");
			return 0;
		}
		finally
		{
			Console.WriteLine("...однако сначала будет выполнен блок
finally...");
			Console.ReadKey(true);
		}
	}
```

3.4.4.5. Оператор throw

Оператор **throw** генерирует исключение. Синтаксис:

```
	<оператор throw> :: throw [<выражение>];
```

Тип выражения должен быть классом System.Exception или его потомком. Подробности приведены в п. 3.4.5.

3.4.4.6. Оператор yield

Оператор **yield** используется для управления итератором (см. пример в п. 4.9.4.2). Синтаксис:

```
	<оператор управления итератором> :: yield return <выражение>;
	<оператор управления итератором> :: yield break;
```

Замечания:

1. Допускается его использование только в методе, который не является анонимным, и тип возвращаемого значения которого реализует интерфейс IEnumerator<T>. Тип выражения должен совпадать с типом T или иметь неявное преобразование к этому типу;

2. Использование оператора **yield** в блоке **finally** влечет ошибку компиляции;

3. Использование оператора **yield return** в блоке **try-catch** влечет ошибку компиляции;

4. Не допускается использование небезопасных блоков (**unsafe**);

5. Параметры итератора не могут иметь модификаторы **ref** или **out**.

3.4.5. Работа с исключительными ситуациями

Одно из основных назначений .NET CLR – недопущение ошибок (что достигается такими средствами, как автоматическое управление памятью и ресурсами в управляемом коде) или хотя бы их обнаружение во время компиляции (благодаря строго типизированной системе). Однако некоторые ошибки можно обнаружить только в период выполнения, а значит, для всех языков, соответствующих спецификации CLS, должен быть предусмотрен единый метод реакции на ошибки. Такой системой обработки ошибок, реализованной в CLR, является обработка *исключительных ситуаций*, или *исключений*. Работа с исключениями нам знакома еще по языку C++, однако, в нем она имеет ряд ограничений:

• Использование исключений не является обязательным. Практически все стандартные функции языка C++ при возникновении ошибки не вызывают исключительных ситуаций, вместо этого они возвращают некоторое специальное значение (код ошибки, нулевой указатель и т.п.) или устанавливают некоторую глобальную переменную (флаг ошибки) в определенное состояние. При этом, если программа не проверяет этот флаг или возвращаемое значение, пользователь может не заметить ошибку. Не заметить исключительную ситуацию нельзя.

• Исключения были разнородными объектами – структура exception в некоторых библиотеках C++, класс System::Exception в библиотеке VCL компании Borland и т.д. В итоге, централизованная обработка ошибок была затруднена.

Исключения – это условия, при которых нормальный ход программы невозможен или нежелателен. Когда вызов метода некоторого класса приводит к ошибке (индекс при работе с массивом не укладывается в диапазон, чтение данных из файла завершено неудачно и т.д.), то мы получаем об этом сообщение в виде исключения. Также мы сами можем генерировать исключительные ситуации и создавать свои собственные классы исключений.

 Пример: Samples\3.4\3_4_5_exception.

3.4.5.1. Класс System.Exception

Все исключения должны иметь тип System.Exception (или производный от него). Полезные для нас члены класса Exception перечислены в табл. 3.22.

Табл. 3.22 – Члены класса System.Exception

Член	Описание
Конструкторы	
`Exception()`	Инициализирует новый экземпляр класса Exception с сообщением по умолчанию
`Exception(`**`string`** `message)`	Инициализирует новый экземпляр класса Exception с указанным сообщением
`Exception(`**`string`** `message, Exception innerException)`	Выполняет инициализацию нового экземпляра класса Exception с указанным сообщением об ошибке и ссылкой на внутреннее исключение, которое стало причиной данного исключения
Методы	
`virtual` `Exception GetBaseException()`	Возвращает исключение, которое является причиной данного исключения
`override string` `ToString()`	Переопределенный метод класс Object, возвращает строковое представление исключения
Поля	
`virtual string` `HelpLink`	Ссылка на файл справки, связанный с этим исключением
`int` `HResult`	Кодированное числовое значение, присвоенное исключению
`Exception InnerException`	Возвращает экземпляр класса Exception, вызвавший текущее исключение.
`virtual string` `Message`	Возвращает сообщение, которое описывает текущее исключение
`virtual string` `Source`	Имя приложения или объекта, вызывавшего ошибку
`virtual string` `StackTrace`	Возвращает строковое представление фрагмента стека вызова в момент возникновения исключения
`MethodBase TargetSite`	Возвращает метаданные метода, вызвавшего исключение

Пример:

```
Exception e1 = new Exception();
Exception e2 = new Exception("Мое сообщение");
```

```
      Exception e3 = new Exception("Мое сообщение", e1);

      Console.WriteLine(e1.Message);
      Console.WriteLine(e2.Message);
      Console.WriteLine(e3.InnerException);

      try
      {
            throw new Exception("Что-то сломалось...");
      }
      catch (Exception e)
      {
            Console.WriteLine("Сообщение: {0}", e.Message);
            Console.WriteLine("Стек вызовов: {0}", e.StackTrace);
            Console.WriteLine("Источник: {0}", e.Source);
      }
```

Вывод на консоль:

```
      Выдано исключение типа "System.Exception".
      Мое сообщение
      System.Exception: Выдано исключение типа "System.Exception".
      Сообщение: Что-то сломалось...
      Стек вызовов: в ExceptionSample.Program.Main() в C:\C#
Samples\3.4\3_4_5_exception\Program.cs: строка 18
      Источник: 3_4_5_exception
```

Класс System.Exception является базовым для большого количества классов исключений. Рассмотрим некоторые из них:

```
      Exception
      ├── SystemException
      │   ├── ArgumentException
      │   │   ├── ArgumentNullException
      │   │   └── ArgumentOutOfRangeException
      │   ├── ArithmeticException
      │   │   ├── DivideByZeroException
      │   │   ├── NotFiniteNumberException
      │   │   └── OverflowException
      │   ├── FormatException
      │   │   └── IO.FileFormatException
      │   ├── IndexOutOfRangeException
      │   ├── InvalidCastException
      │   ├── IOException
      │   │   ├── DirectoryNotFoundException
      │   │   ├── EndOfStreamException
      │   │   ├── FileNotFoundException
      │   │   ├── FileLoadException
      │   │   └── PathTooLongException
      │   ├── NullReferenceException
      │   ├── OutOfMemoryException
      │   │   └── InsufficientMemoryException
      │   ├── RankException
      │   ├── StackOverflowException
      │   └── ...
      └── ...
```

В принципе, их названия говорят сами за себя. Со многими из них мы

уже познакомились, о некоторых других будем говорить далее. Более полную информацию можно получить в библиотеке MSDN.

Большинство унаследованных от System.Exception классов не добавляют функциональности базовому классу. Тогда зачем они были созданы? Причина в том, что один блок **try** может иметь несколько блоков **catch**. Это позволяет программе обрабатывать различные исключения в соответствии с их типом.

3.4.5.2. Основной синтаксис

Обработка исключений

При работе с исключениями, как и в языке C++, используются всего четыре ключевых слова: **try**, **catch**, **finally** и **throw**. Рассмотрим синтаксис операторов блока обработки исключения **try**, **catch** и **finally**:

```
<блок обработки исключения> ::
       try <блокT>
       catch (<тип1> [<идентификатор1>]) <блокC1>
      [catch (<тип2> [<идентификатор2>]) <блокC2>]
      [...]
      [catch (<типN> [<идентификаторN>]) <блокCN>]
      [catch <блокCU>]
      [filally <блокF>]
<блок обработки исключения> ::
       try <блокT>
      [catch (<тип1> [<идентификатор1>]) <блокC1>]
      [catch (<тип2> [<идентификатор2>]) <блокC2>]
      [...]
      [catch (<типN> [<идентификаторN>]) <блокCN>]
       catch <блокCU>
      [filally <блокF>]
<блок обработки исключения> ::
       try <блокT>
       filally <блокF>
```

Замечания:

1. Блок **try** присутствует обязательно. Также должен быть хотя бы один блок **catch** (получаем обработчик исключения типа try-catch), либо блок **finally** (try-finally), либо оба этих вида блоков одновременно (try-catch-finally).

2. Специальные блоки **catch** (C_1, C_2, …, C_N) должны располагаться после блока **try**. Типы данных у идентификаторов должны являться классом System.Exception или его потомками. При этом, <тип$_i$> не должен совпадать ни с одним из использованных ранее типов <тип$_j$> (j < i), а также не должен являться их потомком.

3. Универсальный блок **catch** (C_U), если он описан, должен быть размещен после специальных блоков **catch**, либо, в случае их отсутствия, после блока **try**. Он перехватывает все типы исключений, не обработанных ранее. Поэтому, если выше присутствовал специальный блок **catch**, перехватывающий исключения типа System.Exception, использование универсального блока приведет к предупреждению от компилятора (т.к. он будет недостижим).

4. Блок **finally**, если он описан, должен быть размещен после всех прочих блоков **try** и **catch**.

Схема работы блока обработки исключения приведена на рис. 3.14. Видно, что блок **finally**, если он описан, выполняется и в том случае, когда исключение возникло, и в том, когда не возникло. Соответственно, в него помещается код, который должен быть выполнен в любом случае (закрытие файлов, освобождение ресурсов и т.п.).

Из всех блоков **catch**, если они описаны, выполняется только один – специальный (если найден такой <тип$_i$>, что E является экземпляром этого класса или его потомка) или универсальный (C_i == C_U). Если подходящий специальный блок не найден, а универсальный не описан, либо если исключение в блоке **try** не возникло, управление ни одному из блоков **catch** не передается.

Пример:

```
try { } // Ошибка

try { } // ОК
catch (ArgumentException) { } // ОК
catch (Exception) { } // ОК

try { } // ОК
finally { } // ОК
catch (Exception) { } // Ошибка

try { } // ОК
catch (SystemException) { } // ОК
catch (ArgumentException) { } // Ошибка

try { } // ОК
catch { } // ОК
catch (Exception) { } // Ошибка
finally { } // ОК

try { } // ОК
catch (Exception) { } // ОК
catch { } // Предупреждение
```

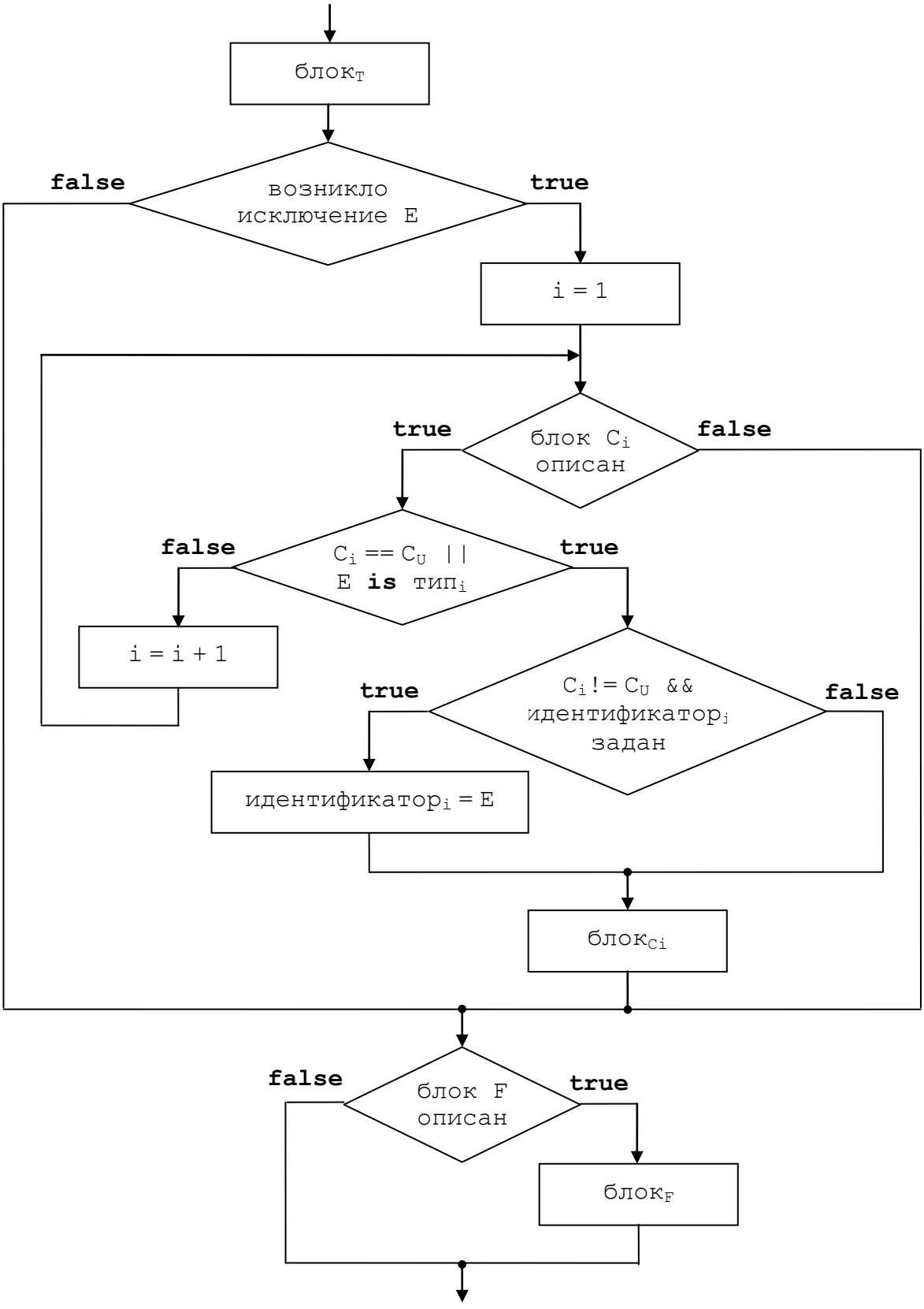


Рис. 3.14 – Схема работы блока обработки исключения

Если в некотором коде возможно появление исключения, но мы не зна-

ем его тип, поступаем следующим образом:

```
try
{
      // какой-то код...
      int[] array = { 1, 2, 3 };
      for (int i = 0; i < 10; i++) array[i] = i;
}
catch (Exception e)
{
      Console.WriteLine(e);
}
```

И видим на консоли класс исключения:

```
      System.IndexOutOfRangeException: Индекс находился вне границ массива.
      в ExceptionSample.Program.Main() в C:\C#
Samples\3.4\3_4_5_exception\Program.cs:строка 7
```

Таким образом, в данном случае возникает исключительная ситуация класса IndexOutOfRangeException.

Вложенные блоки исключений

Блоки обработки исключений могут быть вложенными. Если в процессе выполнения блока **catch** или блока **finally**, в свою очередь, возникает исключение, оно передается в вышестоящий блок обработки исключительных ситуаций. Если таковой не предусмотрен, то исключение обрабатывается компилятором.

Пример:

```
try
{
      try { }
      catch (DivideByZeroException) { }
      finally { }
}
catch (Exception)
{
      try { }
      catch (NullReferenceException) { }
      finally { }
}
finally
{
      try { }
      catch (Exception) { }
      finally { }
}
```

Генерация исключительных ситуаций

Синтаксис оператора, генерирующего исключения (**throw**) был рассмотрен в п. 3.4.4.5:

```
<оператор throw> :: throw [<выражение>];
```

Выражение может быть любым (вызов метода, создание экземпляра класса, идентификатор и т.п.), но его тип должен являться классом System.Exception или его потомком. Пример:

```
SystemException ex = new SystemException ("Что-то сломалось...");
throw ex;
throw new OutOfMemoryException ("Не хватает памяти.");
throw ExceptionMethod("Сообщение");
```

Оператор **throw** без аргументов может использоваться только в блоке **catch**. В этом случае вышестоящему обработчику исключительных ситуаций передается этот же экземпляр исключения E, который привел к передаче управления в данный блок **catch**. Повторная генерация исключений полезна в тех случаях, когда наш код должен предусмотреть какую-то реакцию на ошибку, но вышестоящий обработчик также должен о ней узнать.

Пример:

```
int x;
int a = 1;
int b = 0;

try
{
      try
      {
            x = a / b; // исключение
      }
      catch (DivideByZeroException)
      {
            // чтобы переменная не осталась не инициализированной
            x = 0;
            throw;
      }
}
catch (Exception e)
{
      // внешний обработчик исключения
      Console.WriteLine(e.Message);
}
```

3.4.5.3. Передача исключений из методов

Если при вызове какого-либо метода (принадлежащего текущему или

другому классу) возникает необработанное исключение, то доверять результатам работы этого метода уже нельзя. Все сделанные им изменения в данных необходимо откатить назад.

Другое важное преимущество исключений над другими методами обработки ошибок связано с конструкторами. Так как конструктор не может возвращать значения, простого и понятного способа сигнализации конструктора вызывающему методу об ошибке просто нет. Однако исключения здесь можно использовать, поскольку вызывающий метод требует лишь помещения конструктора в блок **try**.

Пример:

```
      class ExceptionClass
      {
            public int Value;

            public ExceptionClass (int value)
            {
                  Value = 100 / value;
            }

            public int SetValue(int value)
            {
                  return Value = 100 / value;
            }
      }

      static int Main()
      {
            ExceptionClass ec1 = new ExceptionClass(2);

            try
            {
                  ec1.SetValue(0);
            }
            catch
            {
                  Console.WriteLine("Исключение в методе
ExceptionClass.SetValue");
                  ec1.Value = 0;
            }

            ExceptionClass ec2;

            try
            {
                  ec2 = new ExceptionClass(0);
            }
            catch
            {
                  Console.WriteLine("Исключение в конструкторе
ExceptionClass");
                  ec2 = null;
            }
```

```
        return 0;
    }
```

Мы еще не изучили синтаксис описания классов, а также их конструкторов и прочих членов, это будет сделано в главе 4.

3.4.5.4. Получение собственных классов исключений

Создать новый класс исключений достаточно просто – необходимо описать новый класс, наследующийся от одного из имеющихся классов исключений, определить в нем требуемые конструкторы, перегрузить, если это требуется некоторые члены и, опять же, если это требуется, определить новые члены, специфические для нового класса исключений. О синтаксисе описания класса и его членов мы поговорим в главе 4, а пока просто рассмотрим пример:

```
    class MyException : ApplicationException
    {
        public MyException() :
            base("Новое исключение класса MyException") { }
        public MyException(string message) : base(message) { }
        public MyException(string message, Exception innerException) :
            base(message, innerException) { }
    }

    static int Main()
    {
        try
        {
            // некоторый код
            throw new MyException();
        }
        catch (MyException e)
        {
            Console.WriteLine(e.Message);
        }
        catch
        {
            // все остальные исключения
        }

        return 0;
    }
```

§ 3.5. Файловый ввод и вывод

В языке C# существует множество классов для чтения и записи файлов различных форматов – текста, изображений, XML, таблиц баз данных и т.д. Здесь мы рассмотрим только базовые возможности по чтению и записи текстовых и бинарных файлов.

3.5.1. Перечень основных классов файлового ввода-вывода

Классы, используемые в файловом вводе и выводе:

• System.IO.Directory, System.IO.DirectoryInfo – предоставляют методы создания, перемещения и перечисления в директориях и поддиректориях;

• System.IO.DriveInfo – предоставляет методы для доступа к сведениям о диске;

• System.IO.File, System.IO.FileInfo – предоставляют методы создания, копирования, удаления, перемещения и открытия файлов, а также помогает при создании объектов FileStream;

• System.IO.Path – предоставляет методы и свойства для обработки имен каталогов и файлов;

• System.IO.Stream – поддерживает произвольный доступ к потокам данных различной природы, а также синхронные и асинхронные операции ввода-вывода.

Мы не будем рассматривать работу с дисками, каталогами и именами файлов. Необходимые сведения можно найти в библиотеке MSDN. Рассмотрим только операции создания, чтения и записи текстовых и двоичных файлов.

3.5.2. Потоковый ввод и вывод

Базовым классом для большинства других классов потокового ввода-вывода является System.IO.Stream. Это абстрактный класс (см. § 4.6) для любых бинарных (двоичных) потоков данных, т.е. создавать его экземпляры нельзя. Вместо этого создаются экземпляры его потомков, конкретизирующих, куда именно мы хотим записать данные или откуда хотим их считать – из таблицы БД, файла, памяти, сокета HTTP, канала и т.п. Основные его члены перечислены в табл. Д.1 (приложение Д).

Наиболее часто используются следующие его потомки:

• System.IO.Compression.GZipStream – поток, обеспечивающий чтение и запись данных, упакованных в формате GZip;

• System.IO.FileStream – файловые потоки;

• System.IO.MemoryStream – поток, резервным хранилищем которого являются блоки памяти (т.е. позволяет рассматривать блок памяти как бинарный файл для чтения и/или записи).

Их основные члены приведены в табл. Д.2-Д.4 (приложение Д). Как уже было сказано, с помощью данных классов можно обрабатывать потоки любой природы и формата, однако, работать с ними, в силу их универсальности, достаточно сложно. Все данные перед записью необходимо преобразовывать в блоки байтов, а при чтении делать обратное преобразование.

Поэтому для чтения и записи бинарных файлов чаще используются классы System.IO.BinaryReader и System.IO.BinaryWriter (табл. Д.5-Д.6, приложение Д). Они используют в качестве *базового потока* (т.е. потока, реально осуществляющего чтение данных) экземпляр какого-либо потомка класса IO.Stream, но предоставляют более удобные методы для работы с данными более привычных типов – чисел, символов, строк и т.д.

Для чтения и записи текстовых файлов, соответственно, лучше использовать классы, наследуемые от абстрактных классов System.IO.TextReader и System.IO.TextWriter:

• System.IO.StreamReader – считывает символы из потока байтов в определенной кодировке;

• System.IO.StringReader – осуществляет чтение из строки (т.е. позволяет рассматривать строку как текстовый файл для чтения);

• System.IO.StreamWriter – записывает символы в поток байтов в определенной кодировке;

• System.IO.StringWriter – осуществляет запись в строку (т.е. позволяет рассматривать строку как текстовый файл для записи).

Список основных членов данных классов приведен в табл. Д.7-Д.12 (приложение Д).

 Пример: Samples\3.5\3_5_2_stream.

3.5.2.1. Чтение и запись текстового файла

Используем наиболее простой способ работы с текстовыми файлами – функциональность классов IO.StreamWriter и IO.StreamReader.

Пример:

```
StreamWriter sw = new StreamWriter("TextFile.txt");
CultureInfo ukr = new CultureInfo("uk-UA");
DateTime dt = DateTime.Today;
StreamReader sr;
String str;

str = "Дата:";
Console.WriteLine("В файл записана строка: " + str);
sw.WriteLine(str);
str = dt.ToString("dd MMMM yyyy", ukr);
Console.WriteLine("В файл записана строка: " + str);
sw.WriteLine(str);
sw.Close();

Console.WriteLine();
sr = new StreamReader("TextFile.txt");
Console.WriteLine("Размер файла: {0} байт", sr.BaseStream.Length);

while ((str = sr.ReadLine()) != null)
{
      Console.WriteLine("Из файла считана строка: " + str);
}

sr.Close();
```

3.5.2.2. Чтение и запись двоичного файла

Аналогично, используем наиболее простой способ работы с бинарными файлами – функциональность классов IO.BinaryWriter и IO.BinaryReader.

Пример:

```
FileStream fs = new FileStream("BinFile.bin", FileMode.Create);
BinaryWriter bw = new BinaryWriter(fs);
BinaryReader br;
Random rnd = new Random();
int num;

for (int i = 0; i < 10; i++)
{
      num = rnd.Next(-100, 100);
      Console.WriteLine("В файл записано число: " + num);
      bw.Write(num);
}

Console.WriteLine();
bw.Close();
fs.Close();
fs = new FileStream("BinFile.bin", FileMode.Open, FileAccess.Read);
br = new BinaryReader(fs);
Console.WriteLine("Размер файла: {0} байт", fs.Length);

while (fs.Position < fs.Length)
{
      num = br.ReadInt32();
      Console.WriteLine("Из файла считано число: " + num);
```

```
    }

    br.Close();
    fs.Close();
```

3.5.3. Управление ресурсами потока

Потоки при работе с ними выделяют ресурсы. Например, файловые потоки организуют в оперативной памяти буферы. Дело в том, что операции ввода-вывода на физический носитель очень затратные по времени (по сравнению со временем доступа к оперативной памяти, ресурсам процессора и т.д.). Если бы при посимвольной записи текстового файла при каждой операции символ записывался в файл на жестком диске, это происходило бы очень медленно. Чтение происходит несколько быстрее, но все равно недостаточно. Поэтому при записи данные фактически пишутся в некий буфер в ОЗУ, и только при его заполнении попадают на физический носитель. Аналогично при чтении – данные читаются из буфера, а в него считываются с физического носителя блоками. Когда все данные в буфере прочитаны, загружается новый блок. Поэтому так важно закрывать файл по окончании работы (метод Close), особенно при записи – иначе некоторые данные могут остаться в буфере и не попасть в файл.

Другие виды потоков также выделяют различные ресурсы. После закрытия потока они не уничтожаются немедленно. Задача удаления этих ресурсов, как и любых данных в управляемой динамической куче, на которые не осталось ссылок, лежит на сборщике мусора. Но предсказать, когда именно он выполнит свою работу невозможно.

Между тем, представим ситуацию, что только что закончили работу с файлом большого объема. Ресурсы, выделенные потоку, больше не нужны, но еще некоторое неопределенное время будут недоступны для повторного выделения. Либо имеем ситуацию, когда ресурсов в системе и так мало. Если мы уверены, что файл нам больше не нужен, желательно его закрыть и сразу же освободить все занимаемые им ресурсы (которые при большом объеме файла могут быть весьма существенными).

Такая проблема появляется не только при работе с файлами, но и с любыми другими ресурсоемкими задачами. Для ее решения компилятор языка C# позволяет при помощи оператора **using** управлять выделением и освобождением ресурсов. Его синтаксис был приведен в п. 3.4.1.1:

```
      <оператор выделения ресурсов> :: using (<оператор объявления>)
<внедряемый оператор>
```

Обязательное условие – тип данных в операторе объявления должен реализовывать интерфейс System.IDisposable, а все переменные должны быть инициализированы. Все рассмотренные нами в п. 3.5.2 классы реализуют этот интерфейс. Использование оператора **using** гарантирует, что ресурсы будут выделены только на время выполнения внедренного оператора, а по его окончанию будут освобождены.

Пример:

```
      CultureInfo es = new CultureInfo("es-ES");
      DateTime dt = DateTime.Today;
      String str;

      using (StreamWriter sw = new StreamWriter("TextFile.txt"))
      {
            str = "Дата:";
            Console.WriteLine("В файл записана строка: " + str);
            sw.WriteLine(str);
            str = dt.ToString("dd MMMM yyyy", es);
            Console.WriteLine("В файл записана строка: " + str);
            sw.WriteLine(str);
            Console.WriteLine();
      }

      using (StreamReader sr = new StreamReader("TextFile.txt"))
      {
            Console.WriteLine("Размер файла: {0} байт", sr.BaseStream.Length);

            while ((str = sr.ReadLine()) != null)
            {
                  Console.WriteLine("Из файла считана строка: " + str);
            }
      }
```

Здесь переменные sw и sr определены только внутри блоков **using**. Закрывать потоки методом Close не обязательно, метод Dispose, наследуемый классами StreamWriter и StreamReader от интерфейса IDisposable, сделает это автоматически.

 Пример: Samples\3.5\3_5_3_using.

Дополнительные сведения об интерфейсах см. в § 4.9.

3.5.4. Сохранение и загрузка состояния приложения

Часто возникает ситуация, когда состояние отдельных объектов или свойств приложения необходимо восстановить при его следующем запуске. Другими словами, если пользователь закончил сеанс работы с приложением,

при следующем запуске приложение должно восстановить свое состояние (размер и положение окон, настройки интерфейса и т.п.), чтобы пользователь начал работу ровно с того места, на котором закончил ее во время предыдущего сеанса.

Изначально для этой цели использовались *конфигурационные файлы* (обычно с расширением .CFG или .INI). В этих файлах настройки приложения хранились в текстовом виде, примерно следующим образом:

```
[название_секции_1]
название_ключа_1 = значение_1
название_ключа_2 = значение_2
...

[название_секции_2]
...
```

Минус такого подхода состоит в том, что трудно гарантировать целостность данных, хранящихся в текстовом виде. Кроме того, обработка текстовых данных более затратная по времени, чем обработка двоичных данных. Для упорядочения этой информации в ОС Windows 3.1 появился *реестр* – иерархическая база данных параметров и настроек самой ОС Windows, а также установленных приложений и компонентов.

В .NET имеются специальные средства для работы с реестром, в частности, класс System.Microsoft.Win32.Registry. Он позволяет создавать и удалять секции (ветки) и ключи в реестре Windows, читать и записывать значения ключей и т.д.

Однако, слишком большое число записей в реестре приводит к увеличению его размера, а значит, к увеличению потребления ресурсов и временных затрат при обращении к реестру. Поэтому в настоящее время политика .NET сводится к тому, чтобы не записывать настройки приложений в реестр, если в этом нет необходимости. Вместо этого используются такие механизмы, как *параметры приложений* и *сериализация*.

3.5.4.1. Параметры приложения

Итак, параметры приложения – это рекомендованный механизм сохранения настроек для восстановления их при следующих запусках приложения в среде .NET.

Для того, чтобы добавить параметры приложения к проекту, необходимо выбрать пункт меню «Проект» → «Свойства…», и далее – в окне свойств

проекта – вкладку «Параметры». Если приложение еще не имеет параметров, то будет отображена только гиперссылка, при нажатии на которую к проекту будут добавлены три файла – «app.config», «Properties\Settings.settings» и «Properties\Settings.Designer.cs».

Файл «Settings.settings» является XML-файлом с описанием параметров. Среда разработки позволяет редактировать его в удобном виде (рис. 3.15). Для каждого параметра можно ввести его имя, тип, область (пользователя или приложения) и значение по умолчанию.

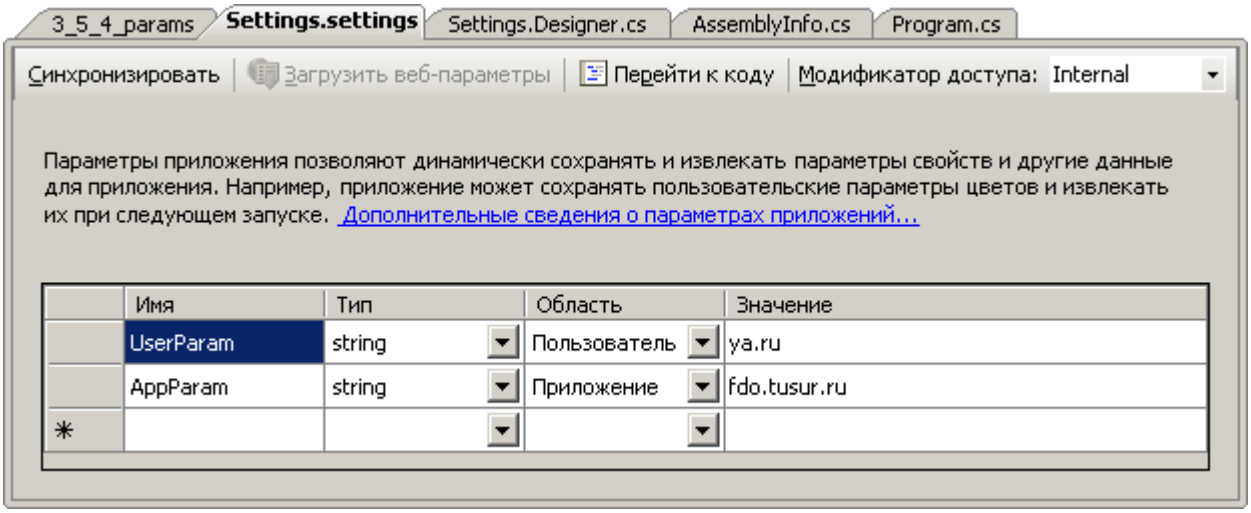


Рис. 3.15 – Редактирование параметров приложения

На рис. 3.15 видно, что к проекту добавлены два параметра – UserParam в области пользователя и AppParam в области приложения. Оба параметра строковые, а значения по умолчанию – это адреса некоторых сайтов. Содержимое файла «Settings.settings» при этом будет следующим:

```
      <?xml version='1.0' encoding='utf-8'?>
      <SettingsFile
xmlns="http://schemas.microsoft.com/VisualStudio/2004/01/settings"
CurrentProfile="(Default)" GeneratedClassNamespace="ParamsSample.Properties"
GeneratedClassName="Settings">
        <Profiles />
        <Settings>
          <Setting Name="UserParam" Type="System.String" Scope="User">
            <Value Profile="(Default)">ya.ru</Value>
          </Setting>
          <Setting Name="AppParam" Type="System.String" Scope="Application">
            <Value Profile="(Default)">fdo.tusur.ru</Value>
          </Setting>
        </Settings>
      </SettingsFile>
```

Файл «Settings.Designer.cs» описывает новый класс приложения, предназначенный для доступа к параметрам. Рассмотрим кратко его основные

конструкции:

```
      ...
      namespace ParamsSample.Properties {
            ...
            internal sealed partial class Settings :
global::System.Configuration.ApplicationSettingsBase {
                  ...
                  public static Settings Default...
                  ...
                  public string UserParam...
                  ...
                  public string AppParam...
            }
      }
```

Итак, в проекте создается новое пространство имен «пространство имен по умолчанию.Properties», в котором описан класс Settings. Не рекомендуется ручное редактирование этого класса. Вместо этого лучше воспользоваться редактором среды разработки (рис. 3.15). Там же задается модификатор доступа этого класса (в данном случае – **internal**). Класс Settings всегда содержит статическое свойство Default. Это экземпляр класса с настройками по умолчанию. В принципе, в приложении можно организовать подборку из нескольких вариантов настроек, и предоставить пользователю возможность выбирать любой из них. Далее описываются свойства, соответствующие ранее созданным параметрам приложения.

Файл «app.config» содержит описание всех параметров приложения. Два рассмотренных выше файла предназначены только для разработчика, а XML-файл «app.config» распространяется вместе с исполняемым файлом приложения. Из него в процессе выполнения приложение получает всю информацию о своих параметрах – имена, типы, значения по умолчанию.

Отличие параметров области приложения от параметров области пользователя в том, что первые доступны только для чтения (это видно даже по коду в файле «Settings.Designer.cs»), в то время как последние можно модифицировать во время работы приложения. После модификации параметры пользователя сохраняются в XML-файле «user.config». Он располагается в папке

```
      <профиль пользователя>\Local Settings\Application Data\<название
компании>\<имя исполняемого файла><суффикс>\<версия приложения>\
```

для ОС Windows XP или

```
      профиль_пользователя\AppData\Local\<название компании>\<имя исполняемого
файла><суффикс>\<версия приложения>\
```

для ОС Windows Vista/7. Название компании берется из настроек ОС, а версия – из информации о сборке (файл «Properties\AssemblyInfo.cs»). Учитывая, что в системе могут быть установлены несколько программ с одинаковыми именами исполняемых файлов, при образовании имени папки к имени исполняемого файла добавляется суффикс, состоящий из строки «_URL_» и дальнейшего набора символов английского алфавита и цифр. Это гарантирует уникальность имени папки с параметрами для каждого приложения.

Рассмотрим пример работы с описанными выше параметрами:

```
      using System;
      using System.Diagnostics;
      using System.IO;
      using ParamsSample.Properties;

      namespace ParamsSample
      {
            class Program
            {
                  static int Main()
                  {
                        ConsoleKeyInfo action;
                        string site = Settings.Default.AppParam;

                        do
                        {
                              Console.Clear();
                              Console.WriteLine("1. Выполнить команду ping для
сайта по умолчанию [{0}]", Settings.Default.AppParam);
                              Console.WriteLine("2. Выполнить команду ping для
сайта [{0}]", Settings.Default.UserParam);
                              Console.WriteLine("3. Изменить адрес сайта");
                              Console.WriteLine("4. Выход");

                              do
                              {
                                    action = Console.ReadKey(true);
                              } while (action.KeyChar < '1' || action.KeyChar
> '4');

                              Console.WriteLine(action.KeyChar);

                              switch (action.KeyChar)
                              {
                                    case '1':
                                          site = Settings.Default.AppParam;
                                          break;

                                    case '2':
                                          site = Settings.Default.UserParam;
                                          break;

                                    case '3':
                                          Console.WriteLine();
                                          Console.WriteLine("Введите адрес:
");
```

```
                                        Settings.Default.UserParam =
Console.ReadLine();
                                        break;
                                }

                                if (action.KeyChar == '1' || action.KeyChar ==
'2')
                                {
                                        ProcessStartInfo psi = new
ProcessStartInfo(Environment.GetFolderPath(Environment.SpecialFolder.System)
+ "\\ping.exe", site);
                                        psi.WindowStyle =
ProcessWindowStyle.Hidden;
                                        psi.RedirectStandardOutput = true;
                                        psi.UseShellExecute = false;
                                        psi.CreateNoWindow = true;
                                        Console.WriteLine("Команда
обрабатывается...");
                                        Process proc = Process.Start(psi);
                                        StreamReader sr = proc.StandardOutput;
                                        Console.WriteLine(sr.ReadToEnd());
                                        proc.WaitForExit();
                                        Console.ReadKey(true);
                                }
                        } while (action.KeyChar != '4');

                        Properties.Settings.Default.Save();
                        return 0;
                }
        }
    }
```

Сначала можно подключить пространство имен, в котором описан класс Settings (чтобы не нужно было каждый раз писать Properties.Settings). Для доступа к параметрам используем конструкцию «Settings.Default.<имя параметра>». Если имя соответствует параметру области пользователя, то такое свойство доступно как для чтения, так и для записи. Параметры области приложения, как уже было сказано, доступны только для чтения. При завершении работы приложения сохраняем параметры вызовом метода Save, который наследуется классом Settings от базового абстрактного класса ApplicationSettingsBase. Другие методы этого класса можно посмотреть в справочной системе.

Данный пример позволяет проверить доступ к сайтам при помощи команды «ping». Адрес одного из сайтов (fdo.tusur.ru) задан как параметр области приложения, поэтому не подлежит модификации. Адрес второго сайта описан в области пользователя, поэтому его можно изменить.

 Пример: Samples\3.5\3_5_4_params.

3.5.4.2. Сериализация

В терминах .NET сериализация – это процесс преобразования экземпляров объектов в поток байтов (т.е. файловый поток вывода). Соответственно, обратный процесс называется *десериализацией* – это преобразование потока байтов (файлового потока ввода) в экземпляры объектов. Потоки ввода-вывода могут быть бинарными, а могут иметь формат XML. Структура XML-файла описывается протоколом SOAP (Simple Object Access Protocol).

Ранее мы говорили о недостатках текстового формата хранения данных. По сравнению с ним, формат XML имеет преимущества – целостность данных в нем обеспечить проще (т.к. имеется определенная иерархическая теговая структура), а за счет оптимизированных средств чтения и записи XML файлов достигается достаточная скорость обработки. Обработка двоичных данных в этом плане еще более надежна и быстра, но теряется наглядность представления данных – в текстовом редакторе изучить их уже не получится. Конечный выбор осуществляет программист на этапе проектирования программной системы.

Для сериализации и десериализации в двоичном формате используется класс System.Runtime.Serialization.Formatters.Binary.BinaryFormatter, а в формате SOAP – System.Runtime.Serialization.Formatters.Soap.SoapFormatter. Есть одна особенность: класс BinaryFormatter содержится в сборке «mscorlib», которая подключается к любому проекту автоматически, а класс SoapFormatter является членом сборки System.Runtime.Serialization.Formatters.Soap, которую нужно вручную подключить к проекту (подробнее о работе со сборками говорится в пункте 4.1.3). Оба класса содержат два наиболее часто используемых метода:

```
void Serialize(Stream serializationStream, object graph);
object Deserialize(Stream serializationStream);
```

Первый метод позволяет осуществлять сериализацию объекта, массива (а также коллекции) объектов или даже *графа объектов*. Под графом объектов подразумевается набор взаимосвязанных между собой при помощи ссылок объектов. Второй метод осуществляет десериализацию объектов. Т.к. результат возвращается в виде ссылки на объект типа **object**, требуется явное преобразование к нужному типу.

Чтобы объект мог участвовать в сериализации, его необходимо пометить атрибутом [Serializable]. В этом случае в сериализации также может

участвовать массив или коллекция из таких объектов. Если же какие-либо члены данного объекта не должны участвовать в сериализации, они помечаются атрибутом [NonSerialized] (больше сведений об атрибутах можно найти в § 5.4).

Пример:

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;
using System.IO;

namespace SerializeSample
{
	class Program
	{
		[Serializable] struct Person
		{
			[NonSerialized] public static int Count = 0;
			public string Name;
			public int Age;
			public Address Address;

			public Person(string name, int age, Address address)
			{
				Name = name;
				Age = age;
				Address = address;
				Count++;
			}
		}

		[Serializable] struct Address
		{
			public string Country;
			public string City;
			public int Index;

			public Address(string country, string city, int index)
			{
				Country = country;
				City = city;
				Index = index;
			}
		}

		static void Out(Person[] a)
		{
			Console.WriteLine("Количество записей: {0}",
Person.Count);
			for (int i = 0; i < a.Length; i++)
			{
				Console.WriteLine("Name[{0}] = {1}", i,
a[i].Name);
				Console.WriteLine("Age[{0}] = {1}", i,
a[i].Age);
				Console.WriteLine("Address[{0}] = {1}, {2},
{3}", i, a[i].Address.Country,
```

```
                                    a[i].Address.Index, a[i].Address.City);
                        }
                  }

                  static int Main()
                  {
                        Person[] m1 = {
                              new Person("Иван", 18, new Address("Россия",
"Томск", 634050)),
                              new Person("Петр", 25, new Address("Россия",
"Новосибирск", 630011)),
                              new Person("Анна", 22, new Address("Россия",
"Кемерово", 650028))
                        };
                        BinaryFormatter bf = new BinaryFormatter();
                        SoapFormatter sf = new SoapFormatter();
                        Person[] m2;
                        ConsoleKeyInfo action;

                        Out(m1);
                        using (FileStream fs = File.Create("person.bin"))
bf.Serialize(fs, m1);
                        using (FileStream fs = File.Create("person.xml"))
sf.Serialize(fs, m1);
                        Console.WriteLine();
                        Console.WriteLine("1. Десериализация из двоичного
потока");
                        Console.WriteLine("2. Десериализация из потока SOAP");

                        do
                        {
                              action = Console.ReadKey(true);
                        } while (action.KeyChar != '1' && action.KeyChar !=
'2');

                        Console.WriteLine(action.KeyChar);
                        Console.WriteLine();

                        if (action.KeyChar == '1') using (FileStream fs =
File.OpenRead("person.bin")) m2 = (Person[])bf.Deserialize(fs);
                        else using (FileStream fs =
File.OpenRead("person.xml")) m2 = (Person[])sf.Deserialize(fs);
                        Out(m2);

                        Console.ReadKey(true);
                        return 0;
                  }
            }
      }
```

В данном примере первое поле структуры Person (Count) не подлежит сериализации. Остальные поля экземпляров этой структуры сохраняются в файлы в обоих доступных форматах. Экземпляры агрегированной структуры Address также сериализуются. Если здесь структуры заменить классами (т.е. от типов по значению перейти к ссылочным типам), то ничего не изменится. В любом случае будет сериализован полный граф объектов.

Далее можно на выбор произвести десериализацию из двоичного файла или из файла в формате SOAP. Как видно по примеру, здесь сериализуется и десериализуется не один экземпляр структуры Person, а сразу массив экземпляров структуры.

В результате выполнения данного кода на консоли отобразится следующая информация:

```
Количество записей: 3
Name[0] = Иван
Age[0] = 18
Address[0] = Россия, 634050, Томск
Name[1] = Петр
Age[1] = 25
Address[1] = Россия, 630011, Новосибирск
Name[2] = Анна
Age[2] = 22
Address[2] = Россия, 650028, Кемерово

1. Десериализация из двоичного потока
2. Десериализация из потока SOAP
2

Количество записей: 3
Name[0] = Иван
Age[0] = 18
Address[0] = Россия, 634050, Томск
Name[1] = Петр
Age[1] = 25
Address[1] = Россия, 630011, Новосибирск
Name[2] = Анна
Age[2] = 22
Address[2] = Россия, 650028, Кемерово
```

 Пример: Samples\3.5\3_5_4_serial.

Здесь можно отметить одну особенность. Хотя были созданы три новых экземпляра структуры Person, счетчик остался равен трем. Это означает, что при десериализации вызов конструктора объекта не производится. Поэтому, если в конструкторе происходит инициализация каких-то полей объекта, не подлежащих сериализации, или вызываются какие-то методы, то эти действия после десериализации придется выполнять самостоятельно, либо реализовать интерфейс System.Runtime.Serialization.ISerializable и описать специальный конструктор:

```
using System.Runtime.Serialization.ISerializable;
...
class|struct <имя_класса_или_структуры> : ISerializable
{
      ...
      private <имя_класса_или_структуры> (SerializationInfo si,
StreamingContext ctx)
```

```
            {
                  ...
            }
            public void GetObjectData(SerializationInfo si, StreamingContext
ctx)
            {
                  ...
            }
            ...
      }
```

Но в этом случае нам придется вручную управлять сериализацией (реализовав метод интерфейса ISerializable.GetObjectData) и десериализацией.

§ 3.6. Директивы препроцессора

Как и в языке C++, в языке C# перед компиляцией код обрабатывается препроцессором – программой, которая ищет в коде специальные директивы и согласно им изменяет код. Некоторые директивы можно определять как извне (с помощью командной строки компилятора или настроек проекта), так и в исходном коде.

3.6.1. Директивы объявлений

Директивы объявления имеют следующий синтаксис:

```
<директива объявления> :: #define <идентификатор>
<директива объявления> :: #undef <идентификатор>
```

Идентификатор должен располагаться на той же строке, что и директива. Можно использовать любой корректный идентификатор, в том числе и ключевые слова языка C#, кроме **true** и **false**. По сравнению с языком C++, на данные директивы налагаются следующие ограничения:

1) Данные директивы могут располагаться только в самом начале файла, до первой лексемы языка;

2) Директива **#define** может объявить только условный символ, но не его значение. Эти символы не могут использоваться для подстановки в текст программы. Их использование ограничено директивами условной компиляции (см. п. 3.6.2).

Для определения условных символов для всего проекта зайдите в его свойства («Проект» → «Свойства…») и найдите на закладке «Построение» поле «Символы условной компиляции» (рис. 3.16).

Вводя в это поле через пробел идентификаторы, мы тем самым делаем новые объявления условных символов. Того же можно добиться, используя опцию компилятора `/define` (краткая форма `/d`) в командной строке:

```
csc.exe /define:<идентификатор1>[;<идентификатор2>...] ...
csc.exe /d:<идентификатор1>[;<идентификатор2>...] ...
```

Если введенный идентификатор будет недопустимым (как «111» на рис. 3.16), компилятор генерирует предупреждение: «Предупреждение: недопустимый параметр для компилятора. "/define:111" будет пропущен».

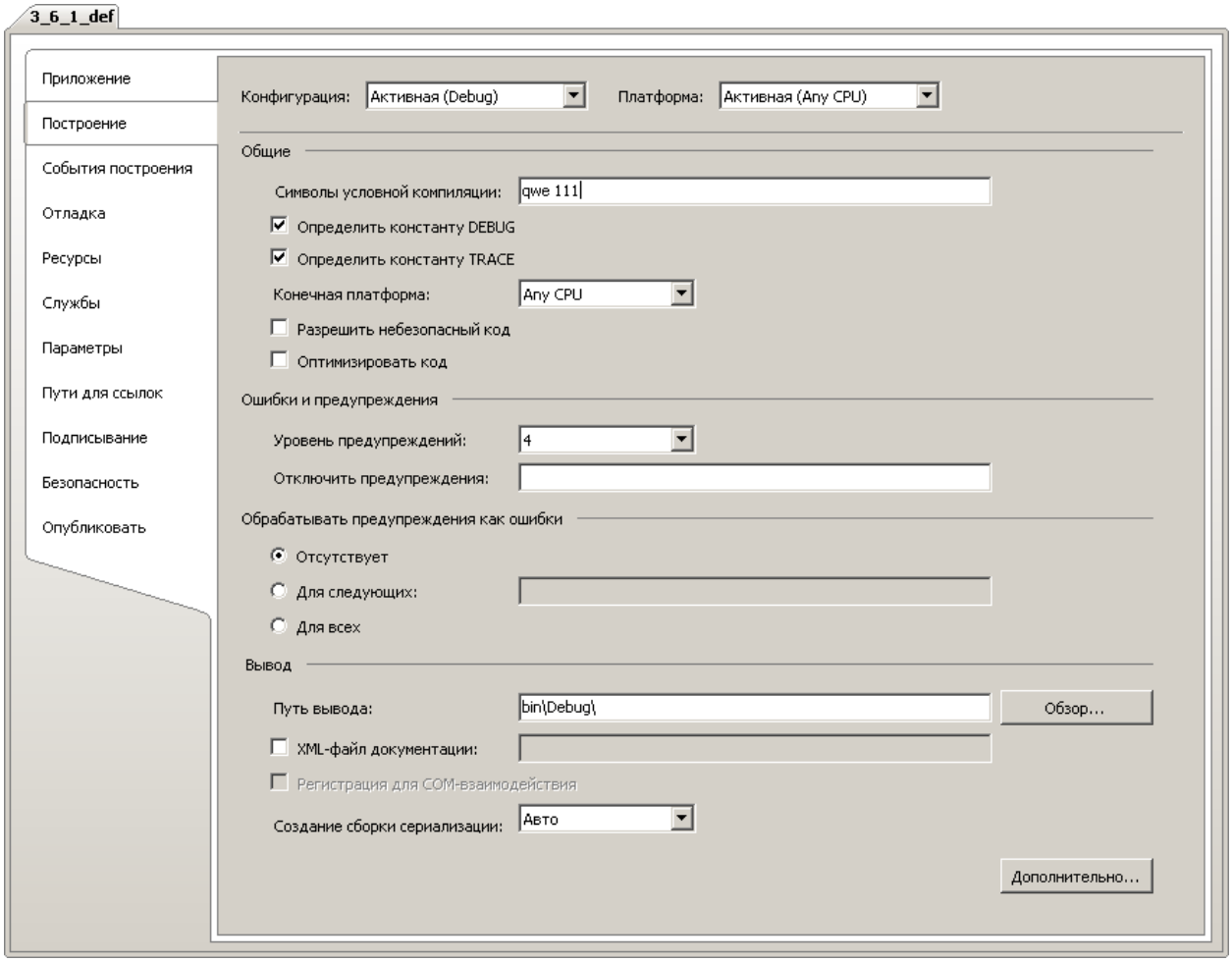


Рис. 3.16 – Директивы объявления в свойствах проекта

Пример:

```
// Ошибка: Требуется идентификатор
// #define 111
// Ошибка: Требуется идентификатор
// #define true
#define for // OK
#define xyz // OK
#undef xyz // OK
#undef abc // OK

using System;

// Ошибка: Невозможно определить символы препроцессора или
// отменить их определение где-либо, кроме начала файла
// #define abc
```

Пример: Samples\3.6\3_6_1_def.

3.6.2. Директивы условной компиляции

Директивы условной компиляции имеют следующий синтаксис:

```
<директива условной компиляции> ::
#if <выражение>
[<лексемы языка и другие директивы>]
[#elif <выражение>
[<лексемы языка и другие директивы>]]
...
[#elif <выражение>
[<лексемы языка и другие директивы>]]
[#else
[<лексемы языка и другие директивы>]]
#endif
```

Данные директивы, как и в языке C++, позволяют исключить некоторые участки кода из компиляции. Если условие директивы **#if** или **#elif** истинно, либо есть директива **#else**, то лексемы, расположенные ниже вплоть до соответствующей директивы **#elif**, **#else** или **#endif**, включаются в компиляцию, иначе исключаются из нее.

Выражение – это логическое выражение, содержащее символы условной компиляции, константы **true** и **false**, а также операции «==», «!=», «&&» и «||». Символ условной компиляции считается истинным, если он определен, и ложным в противном случае. Выражение должно располагаться на одной строке с директивой.

Пример:

```
#define eng
#define rus
#undef eng

using System;

namespace IfSample
{
      class Program
      {
            static int Main()
            {
#if eng // то же, что eng == true
                  Console.WriteLine("Hello, World!");
#endif
#if rus // то же, что rus == true
                  Console.WriteLine("Здравствуй, мир!");
#endif
#if rus == eng // либо два языка, либо ни одного
      #if rus && eng
                  Console.WriteLine("Это многоязычное приложение");
                  Console.WriteLine("This is multilingual application");
      #else
                  Console.WriteLine("???");
      #endif
#else
      #if eng
                  Console.WriteLine("This is monolingual application");
      #elif rus
```

```
                    Console.WriteLine("Это одноязычное приложение");
      #endif
#endif
                    return 0;
              }
      }
}
```

Комбинируя в начале данного кода объявления символов условной компиляции, мы добиваемся изменений в функционировании программы.

 Пример: Samples\3.6\3_6_2_if.

3.6.3. Директивы диагностики

Синтаксис:

```
<директива диагностики> :: #error <сообщение>
<директива диагностики> :: #warning <сообщение>
<директива диагностики> :: #line <индикатор строки>
<индикатор строки> :: <номер строки> "<имя файла>"
<индикатор строки> :: <номер строки>
<индикатор строки> :: default
<индикатор строки> :: hidden
```

Также заимствованные из языка C++ директивы. Первая генерирует ошибку компиляции с указанным сообщением, вторая – предупреждение. Директива **#line** позволяет переопределить имя файла и номер строки, которые будут выводиться в сообщениях компилятора. Опция default в этой директиве восстанавливает оригинальную нумерацию, а hidden скрывает информацию о номерах строк. При отладке такие строки будут пропущены.

Пример:

```
//#undef DEBUG
using System;

namespace DiagSample
{
      class Program
      {
            static int Main()
            {
#line 555 "Главный файл"
#if TRACE
      #warning При запуске данной программы трассировку лучше отключить
#endif
                  throw new Exception();
#line default
#if DEBUG == false
      #error Данная программа должна запускаться в режиме отладки
#endif
                  return 0;
            }
```

```
        }
    }
```

При компиляции мы увидим предупреждение:

```
    #warning: 'При запуске данной программы трассировку лучше отключить'
    Файл c:\C# Samples\3.6\3_6_3_diag\Главный файл
    Строка 556
```

Как видим, директива **#line** повлияла на информацию, выводимую компилятором. Будьте осторожны – при трассировке компилятор попытается найти файл с именем «Главный файл». При запуске программы увидим следующее:

```
    Необработанное исключение: System.Exception: Выдано исключение типа
"System.Exception".
    в DiagSample.Program.Main() в c:\C# Samples\3.6\3_6_3_diag\Главный
файл:строка 558
```

Т.е. директива **#line** влияет также на генерируемые для исключительной ситуации сведения. Если убрать комментарий из первой строки кода, программа не будет компилироваться:

```
    #error: 'Данная программа должна запускаться в режиме отладки'
    Файл C:\C# Samples\3.6\3_6_3_diag\Program.cs   15
    Строка 9
```

Видим, что здесь директивой **#line** default восстановлена оригинальная информация о строках.

 Пример: Samples\3.6\3_6_3_diag.

3.6.4. Директивы регионов

Синтаксис:

```
    <директива региона> ::
    #region [<сообщение>]
    [<лексемы языка и другие директивы>]
    #endregion [<сообщение>]
```

С помощью данных директив можно выделить регион в исходном тексте, Они удобны, так как редактор Visual Studio позволяет сворачивать их. Редактор поддерживает некоторые типы регионов по умолчанию. Регион, который можно свернуть, помечается значком «⊟». Если на него нажать, он свернется в одну строку с многоточием в конце (если на него навести указатель мыши, во всплывающем окне отобразится содержимое региона), а значок изменит свой вид на «⊞». Если на него нажать еще раз, регион снова развернется. К таким регионам по умолчанию относится блок операторов **using**,

пространства имен, классы, методы, комментарии и т.д.

Директивы регионов позволяют создать регион в любом месте. При сворачивании региона будет отображаться сообщение, заданное в директиве **#region**, если оно задано. Сообщение в директиве **#endregion** ни на что не влияет, оно указывается только для повышения читабельности кода.

Пример:

```
for (int i = 0; i < 10; i++)
{
      #region Вывод текста
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      Console.WriteLine("!!!");
      #endregion
}
```

 Пример: Samples\3.6\3_6_4_reg.

3.6.5. Директивы дополнительных опций

Для указания дополнительных опций компилятору используется единственная директива:

```
<дополнительные опции> :: #pragma <тело директивы>
```

Она не влияет на семантику программы. Если тело директивы не распознано, это влечет предупреждение во время компиляции. В настоящий момент компилятор Microsoft поддерживает две дополнительные опции – warning (управление предупреждениями) и checksum (управление контрольными суммами). Мы рассмотрим только первую из них:

```
<управление предупреждениями> ::
#pragma warning <тело предупреждения>
<тело предупреждения> :: disable [<список предупреждений>]
<тело предупреждения> :: restore [<список предупреждений>]
```

Опция disable отключает проверку предупреждений компилятора с указанными номерами (или всех, если список не указан). Опция restore восстанавливает проверку всех или указанных предупреждений до предыдущих значений.

Считается, что хорошая программа должна компилироваться без пре-

дупреждений. Если предупреждения по какой-то причине все же возникают (программа просто еще не дописана и т.п.), от них можно избавиться при помощи этих директив. Например, для следующего кода

```
int i = 0;
if (false && (i++) > 100) i = 0;
```

будет генерироваться следующее предупреждение:

```
Предупреждение: Обнаружен недостижимый код в выражении
Файл C:\C# Samples\3.6\3_6_5_pragma\Program.cs
Строка 11
```

Связано это с тем, что в выражении «`false && (i++) > 100`» код «`i++`» никогда не будет выполнен (см. особенности работы оператора «`&&`»). Хотя, возможно, константа **false** просто заглушка, и в будущем будет заменена другим выражением. Выбираем в окне ошибок это предупреждение и вызываем справочную систему MSDN. Видим описание:

```
Предупреждение компилятора (уровень 4) CS0429
...
```

Итак, мы узнали номер этого предупреждения – 0429 и уровень – 4 (наиболее низкий). Чтобы его отключить, можно использовать опции компилятора. На рис. 3.16 мы видим, что в проекте по умолчанию включен 4-й уровень выдачи предупреждений, самый строгий. Если его понизить хотя бы до 3-го, то предупреждение перестанет появляться. Но тогда мы не увидим всех других предупреждений 4-го уровня во всем проекте! А ведь некоторые из них сигнализируют о потенциальных ошибках. Поэтому можно уровень выдачи предупреждений не изменять, а в поле «Отключить предупреждения» перечислить через пробел номера отключаемых предупреждений. Введем в это поле 0429 – и предупреждение также перестанет генерироваться. Однако, проблема остается прежней – возможно, в других местах программы отсутствие такой проверки приведет к семантической ошибке.

Поэтому используем директиву **#pragma** warning. С ее помощью легко отключить предупреждение 0429 для этой строки, а затем вернуть настройку по умолчанию (номер можно не указывать, т.к. другие предупреждения мы не отключали):

```
// Предупреждение: Нераспознанная директива #pragma
// #pragma Такой директивы нет

class Program
{
      static int Main()
```

```
        {
            int i = 0;
#pragma warning disable 0429
            if (false && (i++) > 100) i = 0;
#pragma warning restore
            return 0;
        }
}
```

Данные опции для всего проекта можно изменить также из командной строки компилятора:

```
csc.exe /warn:<уровень> ...
csc.exe /nowarn:<номер1>[,<номер2>...] ...
```

 Пример: Samples\3.6\3_6_5_warn.

4. КЛАССЫ И ИНТЕРФЕЙСЫ

Классы – сердце каждого объектно-ориентированного языка. Класс представляет собой инкапсуляцию данных и методов их обработки. В том, что касается классов и многих функций языка, C# кое-что заимствует из C++, но также привносит и кое-что новое.

Также мы рассмотрим делегаты и интерфейсы. Но сначала поговорим о пространствах имен, включающих в себя описания других объектов.

§ 4.1. Пространства имен

Программы на языке C# организованы с помощью пространств имен. Пространства имен используются как в качестве «внутренней» системы организации для программы, так и в качестве «внешней» системы организации – способа представления программных элементов другим программам.

В целом, структура исходного файла .CS может быть следующей:

```
<исходный файл> ::
      [<директивы объявлений>]
      [<внешние псевдонимы>]
      [<директивы использования>]
      [<глобальные атрибуты>]
      [<члены пространства имен>]

<члены пространства имен> :: <член пространства имен> [...]

<член пространства имен> :: <пространство имен>
<член пространства имен> :: [<атрибуты>] [<модификатор>] <перечисление>
<член пространства имен> :: [<атрибуты>] [<модификатор>] <структура>
<член пространства имен> :: [<атрибуты>] [<модификатор>] <класс>
<член пространства имен> :: [<атрибуты>] [<модификатор>] <делегат>
<член пространства имен> :: [<атрибуты>] [<модификатор>] <интерфейс>

<модификатор> :: internal
<модификатор> :: public
```

Итак, в исходном файле могут быть описаны:

1. Директивы объявлений (§ 3.6);
2. Внешние псевдонимы (п. 4.1.3);
3. Директивы использования (п. 4.1.2);
4. Глобальные атрибуты (§ 5.4);
5. Члены пространства имен – вложенные пространства имен, перечисления и структуры (§ 3.1), классы (§ 4.2), делегаты (§ 4.8), а также интерфейсы (§ 4.9).

Для членов пространства имен доступны два модификатора – **internal**

(член доступен только в пределах текущей программы) и **public** (неограниченный доступ). По умолчанию используется модификатор **internal**. Модификаторы доступа, разрешенные в других конструкциях программы, рассмотрены в § 4.2. Атрибуты рассмотрим в § 5.4.

4.1.1. Описание пространства имен

Синтаксис описания пространства имен в языке C#:

```
<пространство имен> :: namespace <имя пространства имен>
"{"
      [<члены пространства имен>]
"}"

<имя пространства имен> :: <идентификатор>
<имя пространства имен> :: <идентификатор>.<имя пространства имен>
```

Как мы видим, он соответствует синтаксису языка C++, только имя пространства может состоять из нескольких идентификаторов, разделенных точкой. Запись

```
namespace <имя1>.<имя2>
{
      // члены
}
```

соответствует записи

```
namespace <имя1>
{
      namespace <имя2>
      {
            // члены
      }
}
```

Полным именем члена пространства имен будет

```
<имя пространства имен>.<имя члена>
```

Пространство с одинаковым именем может объявляться несколько раз. В этом случае все члены, описанные во всех объявлениях пространства, считаются членом одного объединенного пространства имен. Элементы вложенного пространства имен нельзя использовать как элементы вмещающего пространства. Имя вложенного пространства при этом должно быть обязательно указано. А пространства с одинаковыми именами, но находящиеся в разных ветвях иерархии, не считаются одним целым. Пример:

```
namespace Test1
{
```

```
            struct S1 { }
            namespace Test2
            {
                  struct S2 { }
            }
      }

      namespace Test2.Inner
      {
            struct S3 { }
      }

      namespace Test2.Outer
      {
            struct S4 { }
      }

      namespace Test1
      {
            struct S5 { }
      }

      namespace Test2
      {
            struct S6 { }
            public struct S7 { }
            internal struct S8 { }
            private struct S9 { } // Ошибка
      }

      namespace NamespaceSample
      {
            class Program
            {
                  static int Main()
                  {
                        Test1.S1 s1;
                        Test1.Test2.S2 s2;
                        Test2.Inner.S3 s3;
                        Test2.Outer.S4 s4;
                        Test1.S5 s5;
                        Test2.S6 s6;

                        Test2.S2 ss2; // Ошибка
                        Test2.S3 ss3; // Ошибка
                        return 0;
                  }
            }
      }
```

Первая ошибка связана с тем, что используется неразрешенный модификатор доступа. Вторая – с тем, что структура S2 объявлена в пространстве имен Test1.Test2, а не Test2. Третья – структура S3 объявлена в пространстве имен Test2.Inner, а не Test2.

Пространства имен следует использовать для улучшения структуры программы. Все классы, близкие по функциональности, помещаются в от-

дельное пространство имен. В свою очередь, пространства имен образуют иерархию, отражающую классификацию содержащихся в них элементов.

 Пример: Samples\4.1\4_1_1_namespace.

4.1.2. Директивы использования

Для работы с элементами пространства имен предназначена директива использования **using**. Причем в языке C++ мы можем использовать как пространство имен целиком, так и отдельные его элементы, например:

```
namespace X
{
	typedef int Type1;
	namespace Y
	{
		typedef int Type2;
	}
}

void main(void)
{
	{
		Type1 z1; // Ошибка – Type2 не описан
		X::Type1 z2; // OK
		X::Y::Type2 z3; // OK
	}
	{
		using namespace Y; // Ошибка – неизвестное пространство Y
		Type2 z; // Ошибка – Type2 не описан
	}
	{
		using namespace X; // Используем все элементы X
		Type1 z1; // OK
		Type2 z2; // Ошибка – вложенные пространства не используются
		Y::Type2 z3; // OK
	}
	{
		using namespace X::Y; // Используем все элементы X::Y
		Type1 z1; // Ошибка – мы использовали только Y, но не X
		Type2 z2; // OK
		X::Type1 z3; // OK
	}
	{
		using namespace X; // Используем все элементы X
		using namespace Y; // Используем все элементы Y
		Type1 z1; // OK
		Type2 z2; // OK
	}
	{
		using X::Type1; // Используем только X::Type1
		using Y::Type2; // Ошибка – неизвестное пространство Y
		using; // Используем только X::Y::Type2
		Type1 z1; // OK
		Type2 z2; // OK
	}
```

```
}
```

Примерно те же правила используются и в языке C#. Использовать можно как пространство имен целиком, так и отдельные его элементы (задавая псевдонимы). Использование вложенных пространств имен нужно указывать явно, т.к. рекурсивное включение не поддерживается.

 Пример: Samples\4.1\4_1_2_using.

4.1.2.1. Директива импортирования

Синтаксис использования директивы импортирования:

```
using <имя пространства имен>;
```

В отличие от языка C++, ключевое слово **namespace** не указывается, а все директивы импортирования должны быть описаны до членов пространства имен. Такой синтаксис позволяет подключать только пространства имен целиком. После этого можно использовать имена членов пространства, не указывая предварительно имя пространства.

Рассмотрим варианты использования двух интерфейсов – IList из пространства имен System.Collections и IList<T> из System.Collections.Generic (см. п. 4.9.4). Пример 1:

```
using System;

namespace UsingSample
{
	class Program
	{
		static int Main()
		{
			IList<int> list1 = null; // Ошибка
			Collections.IList list2 = null; // Ошибка
			System.Collections.Generic.IList<int> list3 = null;
			System.Collections.IList list4 = null;

			return 0;
		}
	}
}
```

Итак, директивы импортирования не являются рекурсивными. Хотя мы используем пространство имен System, это не означает использование вложенных пространств System.Collections и System.Collections.Generic. С этим связана первая ошибка. Вторая ошибка не проявилась бы в языке C++, т.к. в нем можно использовать относительные имена пространств (см. в примере

выше – указав директиву использования X, мы можем использовать элементы пространства Y, указывая Y::<имя элемента>). В языке C# имя пространства либо не указывается вообще, либо указывается полностью.

Пример 2:

```
namespace UsingSample2
{
	using System.Collections;

	class MyClass2
	{
		static void F2()
		{
			IList<int> list1 = null; // Ошибка
			IList list2 = null; // OK
			System.Collections.Generic.IList<int> list3 = null;
			System.Collections.IList list4 = null;
		}
	}
}
```

Теперь компилятор считает, что IList нельзя использовать как универсальный тип. В используемом пространстве имен System.Collections интерфейс IList описан, но его универсальная версия описана в пространстве System.Collections.Generic, для которого директива использования не приведена. Полные имена интерфейсов можно использовать в обоих случаях.

4.1.2.2. Внутренние псевдонимы

Синтаксис описания псевдонима:

```
<внутренний псевдоним> ::
	using <идентификатор> = <имя пространства имен>;
	using <идентификатор> = <имя пространства имен>.<имя члена>;
```

После чего вместо имени пространства или члена пространства можно использовать указанный идентификатор. Пример:

```
namespace UsingSample3
{
	using Coll = System.Collections;
	using List = System.Collections.Generic.IList<int>;

	class MyClass3
	{
		static void F3()
		{
			List list1 = null;
			Coll.IList list2 = null;
			Coll.Generic.IList<int> list3 = null;
		}
	}
```

```
    }
```

Теперь для обозначения интерфейса IList можно использовать запись Coll.IList, а для IList<**int**> – псевдоним List или запись Coll.Generic.IList<**int**>. Для доступа к элементам псевдонима можно также использовать оператор «::»:

```
    Coll::IList list4 = null;
```

4.1.3. Ссылки на сборки

Сборка – это основной строительный блок приложения .NET Framework. Все управляемые типы и ресурсы содержатся в сборках и помечаются либо как доступные только в пределах сборки (**internal**), либо как доступные из кода в других сборках (**public**). Сборки также играют ключевую роль в системе безопасности. Управление доступом для кода использует сведения о сборке для определения набора разрешений, предоставляемых коду в сборке.

4.1.3.1. Просмотр информации о сборках

Некоторые приемы работы со сборками мы рассмотрим в § 5.3. Посмотреть список сборок, подключенных к проекту, можно в обозревателе решений среды MS Visual Studio 2008 (рис. 4.1).

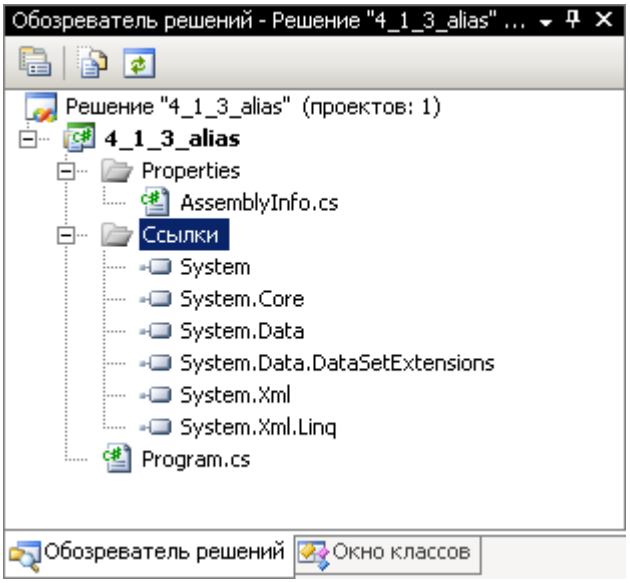


Рис. 4.1 – Ссылки на сборки

Не следует путать понятия сборки и пространства имен. Например, ссылка System – это не синоним пространства имен System, а пространства

имен System.Core вообще не существует. Сборки и пространства имен имеют отношение «многие ко многим». В одной сборке могут быть описаны объекты различных пространств, а объекты одного пространства могут находиться в разных сборках.

Чтобы посмотреть, какие объекты включены в сборку, выберите ее в списке ссылок, нажмите правую кнопку мыши и выберите пункт контекстного меню «Просмотр в обозревателе сборок» (рис. 4.2).

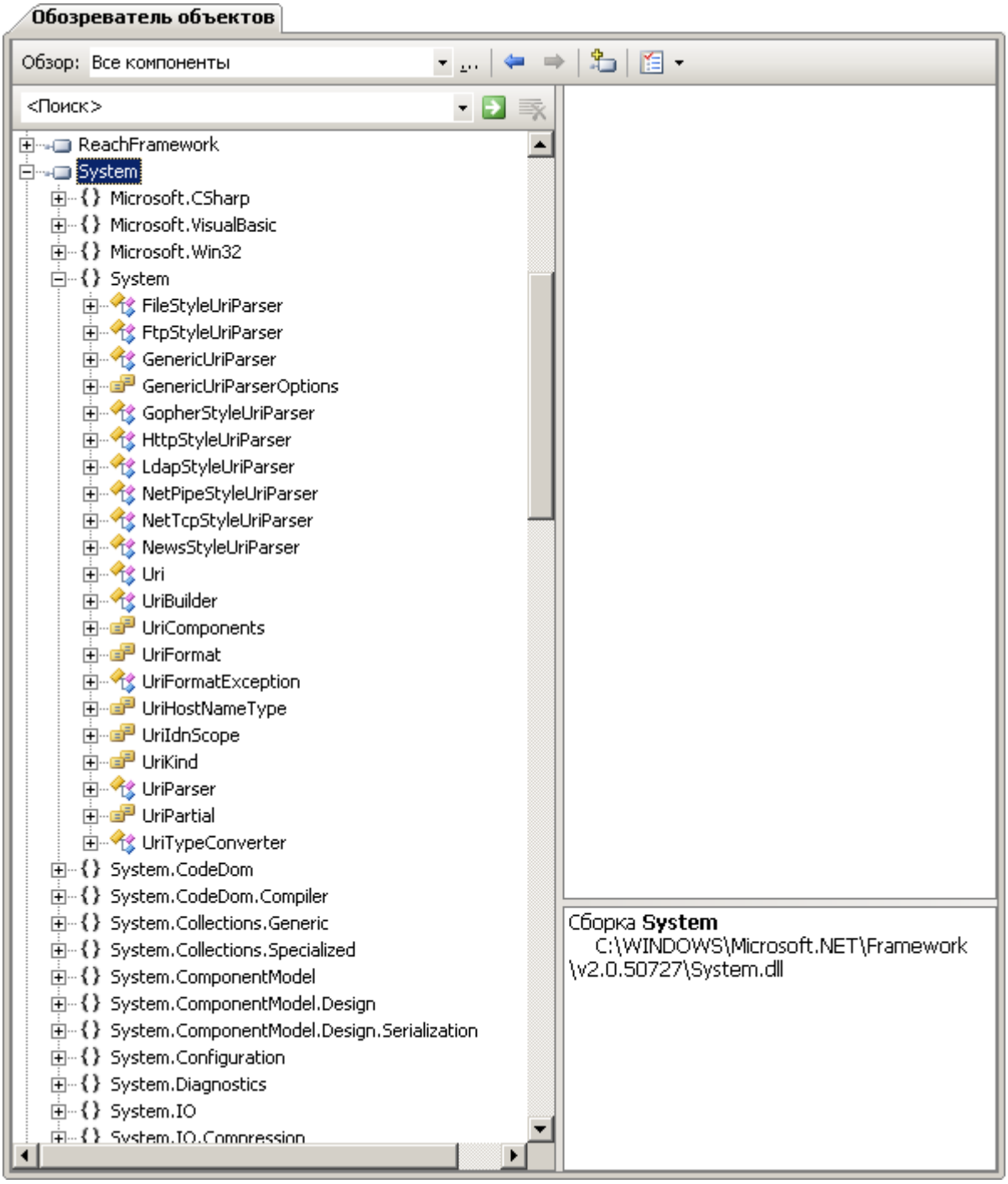


Рис. 4.2 – Обозреватель сборок

Попытка использовать объекты, не описанные в одной из подключенных сборок, приведет к ошибке компиляции «Не удалось найти имя типа или пространства имен "…" (пропущена директива using или ссылка на сборку?)». Пример:

```
System.Security.Cryptography.ProtectedData data = null; // Ошибка
```

Класс ProtectedData включен в сборку System.Security, которая по умолчанию к проекту не подключается. Если это сделать вручную (выбрав пункт меню «Проект» → «Добавить ссылку…» или посредством контекстного меню обозревателя решения), то ошибка исчезнет (рис. 4.3).

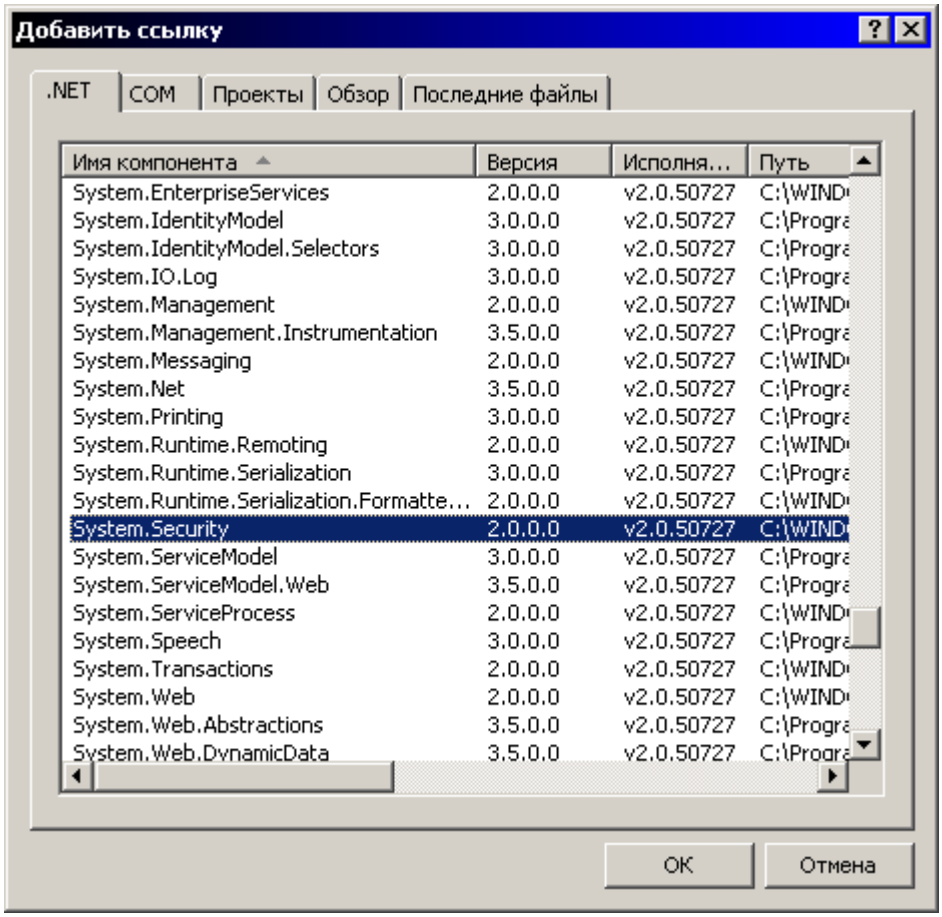


Рис. 4.3 – Добавление ссылки на сборку

Интересно, что класс Console не входит в сборку System. Поиск показывает, что он определен в сборке «mscorlib». Это базовая сборка, в которой определены классы базовых типов данных и некоторые другие основные классы.

Также можно добавлять ссылки на сборки, используя командную стро-

ку компилятора. Параметр /reference (краткая форма /r) указывает компилятору C# импортировать сведения типа **public** из указанного файла сборки в текущий проект. Синтаксис:

```
csc.exe /reference:[<псевдоним>=]<имя файла> ...
csc.exe /reference:<имя файла> ...
```

4.1.3.2. Внешние псевдонимы

Параметр /reference позволяет задавать псевдоним сборки, однако, подключая стандартные сборки, компилятор по умолчанию этого не делает (рис. 4.4). Как видно в окне вывода данных о построении проекта, ни один псевдоним не указан.

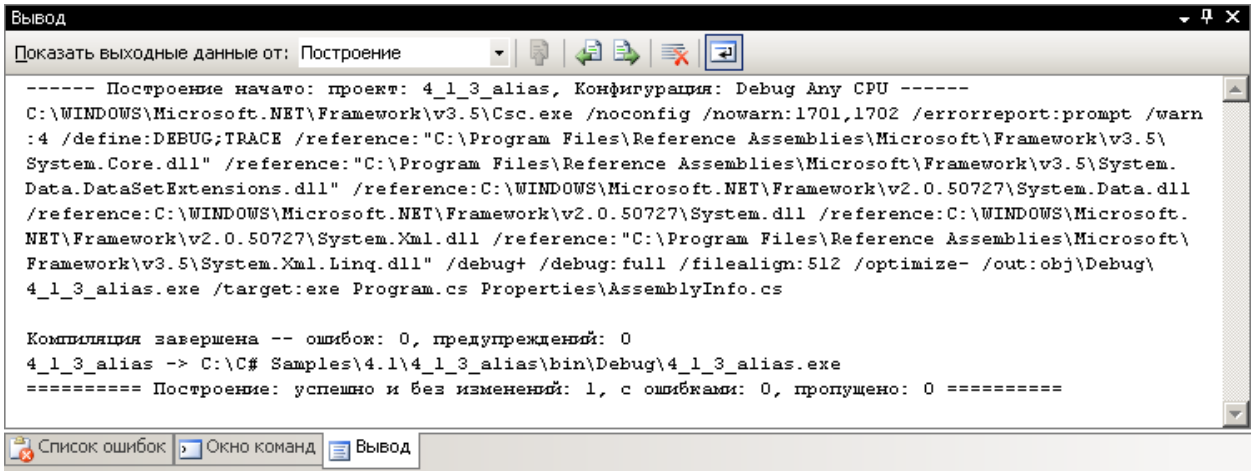


Рис. 4.4 – Протокол построения проекта

Это можно сделать самостоятельно в окне свойств (рис. 4.5). После этого синтаксис использования псевдонимов в тексте программы будет следующим:

```
extern alias <псевдоним>;
```

Тогда можно обращаться к элементам сборки через псевдоним:

```
<псевдоним>::<элемент сборки>
```

Например, зададим сборке System псевдоним «X». Теперь корректен следующий код:

```
extern alias X;

using System;

namespace AliasSample
{
      class Program
```

```
        {
            static int Main()
            {
                X::System.Collections.Generic.Stack<int> stack;

                return 0;
            }
        }
}
```

Это может потребоваться в случае использования нестандартных сборок или при конфликтах имен элементов сборок.

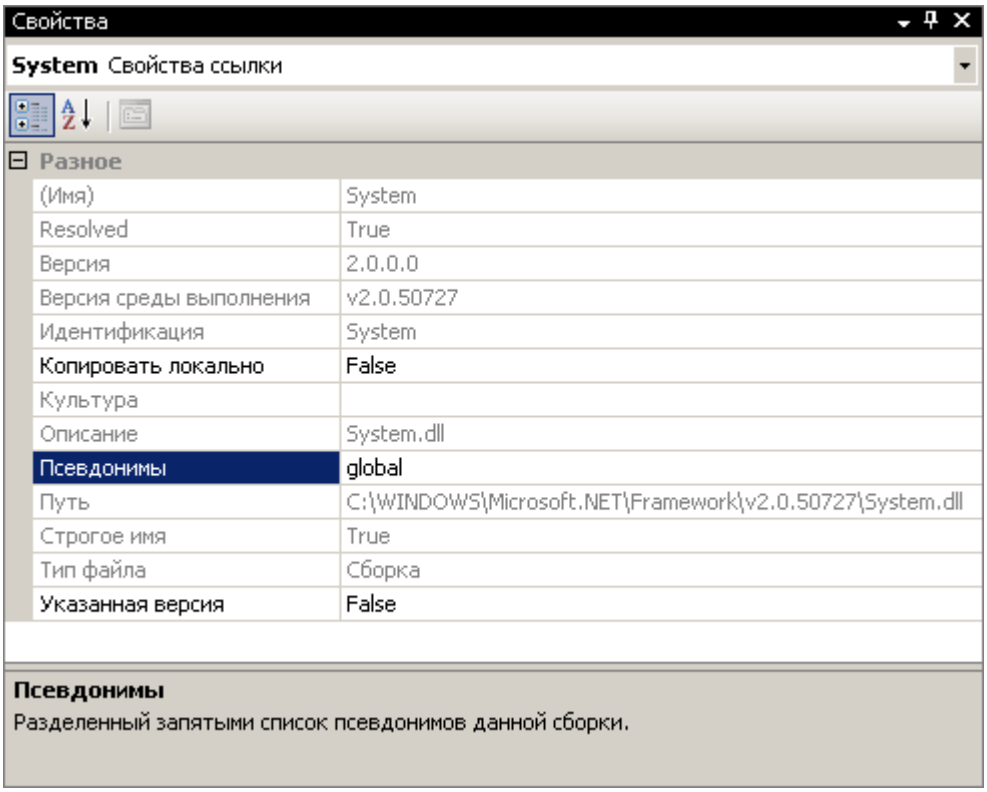


Рис. 4.5 – Окно свойств

Как мы видим на рис. 4.5, все сборки по умолчанию имеют псевдоним, которому соответствует ключевое слово **global**. Его можно использовать как обычный псевдоним. Все объекты, объявленные вне пространства имен, и все пространства имен верхнего уровня входят в глобальное безымянное пространство имен, имеющее псевдоним **global**:

```
struct S { }

namespace AliasSample
{
    class Program
    {
        static int Main()
        {
```

```
                        global::System.Collections.Generic.IList<int> list;
                        global::AliasSample.Program prog;
                        global::S s;

                        return 0;
                    }
                }
            }
```

 Пример: Samples\4.1\4_1_3_alias.

§ 4.2. Описание класса

Многое, сказанное в этой главе, можно также отнести и к структурам. Мы помним, что структуры, в отличие от классов, являются типами по значению и не допускают наследование от других структур и классов. Однако, в том, что касается реализации интерфейсов и описания членов класса, между классами и структурами много общего. Поэтому, рассказывая о синтаксисе описания тех или иных конструкций в классах, мы будем отмечать ограничения, накладываемые при использовании данных конструкций в структурах, если таковые имеются.

Синтаксис описания класса следующий:

```
[<атрибуты>] [<модификаторы>] class <идентификатор> [<параметры типа>]
      [: <список наследования>] [<ограничения параметров типа>]
"{"
      [<тело класса>]
"}" [;]
```

Итак, в описании класса участвуют следующие конструкции:

- атрибуты (см. § 5.4);
- модификаторы (п. 4.2.1);
- параметры типа (§ 5.1);
- список наследования (§ 4.6);
- ограничения параметров типа (§ 5.1);
- тело класса (п. 4.2.2).

4.2.1. Модификаторы класса

Имеющиеся модификаторы классов доступа перечислены в табл. 4.1.

Табл. 4.1 – Модификаторы доступа в языке C#

Модификатор	Описание
abstract	Объявляет *абстрактный* класс, экземпляры которого создавать запрещено. Не может быть запечатанным (**sealed**)
internal	Доступ к классу не ограничен в пределах данной программы
new	Допускается только во вложенных классах. Он определяет сокрытие классом унаследованного члена с таким же именем

partial	Объявляет *разделяемый класс*, отдельные части которого могут быть описаны в разных местах программы
private	Допускается только во вложенных классах. Доступ к классу предоставляется только для класса, в котором он описан
protected	Допускается только во вложенных классах. Доступ к классу предоставляется только для класса, в котором он описан, и для его потомков
protected internal	Допускается только во вложенных классах. Доступ к классу не ограничен в пределах данной программы, в противном случае предоставляется только для класса, в котором он описан, и его потомков
public	Доступ к классу не ограничен
sealed	Объявляет *запечатанный класс*, для которого запрещено создание производных классов
static	Объявляет *статический класс*. Запрещено создавать его экземпляры, а все члены должны быть статическими. Не может быть запечатанным или абстрактным, не может наследоваться от других классов
unsafe	Весь код, описанный в классе, является небезопасным.

Пример:

```
abstract class AbsCls { } // OK
abstract sealed class AbsSealed { } // Ошибка

new class XXX { } // Ошибка

class ZZZ1
{
      public class XXX { }
      public class YYY { }
}

class ZZZ2 : ZZZ1
{
      public class XXX { } // Предупреждение
      public new class YYY { } // OK
}

public class Pub { } // OK
protected class Prot { } // Ошибка
private class Priv { } // Ошибка
internal class Int { } // OK
protected internal class PInt { } // Ошибка
```

```
class Cls
{
      public class Pub { }
      protected class Prot { }
      private class Priv { }
      internal class Int { }
      protected internal class PInt { }
}

sealed class QQQ { } // OK
class WWW : QQQ { } // Ошибка

static abstract class SAC { } // Ошибка
static sealed class SSC { } // Ошибка
static class SC : Cls { } // Ошибка

static class StatClass
{
      public void Pub() { } // Ошибка
      protected static void Prot() { } // Ошибка
      public static void Met() { } // OK
      public static StatClass() { } // Ошибка
      static StatClass() { } // OK
}

class Program
{
      static int Main()
      {
            AbsCls abs = new AbsCls(); // Ошибка

            Pub      pub1  = new Pub(); // OK
            Int      int1  = new Int(); // OK
            Cls.Pub  cpub  = new Cls.Pub(); // OK
            Cls.Prot cprot = new Cls.Prot(); // Ошибка
            Cls.Priv tpriv = new Cls.Priv(); // Ошибка
            Cls.Int  tint  = new Cls.Int(); // OK
            Cls.PInt tpint = new Cls.PInt(); // OK
            StatClass stat = new StatClass(); // Ошибка

            StatClass.Met(); // OK
            return 0;
      }
}
```

Как мы уже говорили, для классов, объявленных в пространстве имен, по умолчанию подразумевается модификатор доступа **internal**. Для классов, объявленных в теле другого класса – **private**.

Части разделяемого класса могут быть описаны по отдельности, и даже размещены в разных файлах. Пример:

```
partial class PartCls
{
      public void X() { }
}

partial class PartCls
```

```
{
      public void Y() { }
}

class Program
{
      static int Main()
      {
            PartCls part = new PartCls();
            part.X(); // OK
            part.Y(); // OK
            return 0;
      }
}
```

Для структур запрещены модификаторы **abstract**, **sealed** и **static**, все остальное верно и для них.

 Пример: Samples\4.2\4_2_1_modifiers.

4.2.2. Члены класса

Объявление класса может содержать объявления:

1. типов (п. 4.2.5);
2. констант (п. 4.3.1);
3. полей (п. 4.3.2);
4. методов (§ 4.4);
5. конструкторов экземпляров (п. 4.4.2);
6. статических конструкторов (п. 4.4.2);
7. деструкторов (п. 4.4.3);
8. свойств (п. 4.5.1);
9. индексаторов (п. 4.5.2);
10. операторов (п. 4.7.3);
11. событий (§ 4.8);

Общий синтаксис описания членов класса следующий:

```
[<атрибуты>] [<модификаторы>] <описание>
```

Модификаторы, использующиеся при описании членов класса, перечислены в табл. 4.2. В последнем столбце указано, для каких членов применим данный модификатор.

Табл. 4.2 – Модификаторы членов класса

Модификатор	Описание	Члены
abstract	Абстрактный член	4,8,9,11

const	Член является константой	2(*)
event	Описывает событие	11(*)
extern	Член имеет внешнюю реализацию	4-11
internal	Член доступен во всей программе	1-5,8,9,11
new	Сокрытие унаследованного члена	1-4,8,9,11
override	Перегрузка унаследованного члена	4,8,9,11
partial	Разделяемый метод класса	4
protected	Член доступен в классе и его потомках	1-5,8,9,11
private	Член доступен только в классе	1-5,8,9,11
public	Член доступен отовсюду	1-5,8,9,10(*),11
readonly	Поле только для чтения	3
sealed	Запрет дальнейшей перегрузки члена	1(**),4,8,9,11
static	Статический член	1(**),3,4,6(*),8,10(*),11
unsafe	Небезопасный контекст	1(***),3-11
virtual	Виртуальный член	4,8,9,11
volatile	Поле, не проходящее оптимизацию	3

Примечания: (*) – обязательное условие; (**) – если вложенный тип описывает класс; (***) – если вложенный тип не является перечислением.

Абстрактный член в классе имеет описание, но не имеет реализации. Т.е. класс, содержащий абстрактный член, является *незавершенным*. Он также должен быть абстрактным. Абстрактные члены предназначены для перегрузки классами-потомками.

С локальными переменными-константами мы уже знакомились в § 3.1, в классе также могут быть объявлены константы-члены класса.

Модификатор **extern** указывает, что реализация члена класса находится вне программы, обычно в DLL (см. § 5.5) или внешней сборке. Его нельзя использовать совместно с модификатором **abstract**.

О модификаторах доступа **public**, **internal**, **protected** и **private** мы уже говорили, все сказанное для классов верно и для их членов. Аналогично для модификатора **new**.

Модификатор **virtual** применяется для описания виртуального члена. Виртуальный член может быть перегружен в классе-потомке. От абстрактного члена он отличается тем, что имеет реализацию (тело). Модификатор **override** используется для перегрузки абстрактных и виртуальных членов в

классах-потомках. Модификатор **sealed** запрещает дальнейшую перегрузку в классах-потомках виртуальных и абстрактных методов, объявленных в одном из классов-предков данного класса.

Метод с модификатором **partial** может содержать одно отдельное определяющее объявление (не имеющее тела) и одну реализацию. Если реализация не будет найдена, такой метод будет исключен из компиляции. Обычно такие методы применяются в разделяемых классах, когда тело метода описано в одной части класса, а реализация – в другой.

Поле с модификатором **readonly** доступно только для чтения. Модификатор **volatile** говорит компилятору, что содержимое поля может изменяться несколькими потоками, поэтому не нужно пытаться оптимизировать доступ к нему.

Для членов структур запрещены модификаторы **abstract**, **protected**, **sealed**, **virtual**.

Некоторые имена и сигнатуры резервируются компилятором языка C# и не могут быть использованы для членов класса:

1) Имена, совпадающие по написанию с ключевыми словами языка (если они не предваряются символом «@», см. п. 3.1.6.1);

2) Имя класса. Только имя конструктора и деструктора может совпадать с именем класса (см. п. 4.4.2, 4.4.3);

3) Для свойства P типа T зарезервированы сигнатуры «T get_P()» и «**void** set_P(T value)» (см. п. 4.5.1);

4) Для индексатора типа T со списком формальных параметров L зарезервированы сигнатуры «T get_Item(L)» и «**void** set_Item(L, T value)» (см. п. 4.5.2);

5) Для события E типа делегата T зарезервированы сигнатуры «**void** add_E(T handler)» и «**void** remove_E(T handler)» (см. § 4.8);

6) Для класса, содержащего деструктор, зарезервирована сигнатура «**void** Finalize()» (см. п. 4.4.3).

Под сигнатурой понимается комплексное описание члена. Как будет показано ниже, методы или индексаторы могут иметь совпадающие имена, если они различаются списком формальных параметров. Пример:

```
class Program
{
      int if; // Ошибка
      int @if; // ОК
      int Program; // Ошибка
```

```
        int get_A;
        int set_A;
        int A { get; set; } // Ошибка

        void get_B(int i) { }
        void set_B(int i, int j) { }
        int B { get; set; } // ОК

        ~Program() { }
        int Finalize; // Ошибка
    }
```

 Пример: Samples\4.2\4_2_2_members.

4.2.3. Статические члены и члены экземпляров

Отдельно остановимся на статических классах и членах класса. Мы можем определить член класса как статический (static member) или член экземпляра (instance member).

По умолчанию каждый член определен как член экземпляра. Это значит, что для каждого экземпляра класса делается своя копия этого члена. Для члена экземпляра определен идентификатор **this**, являющийся ссылкой на экземпляр, для которого в настоящий момент используется данный член. Например:

```
    class MyTest
    {
        private int x;

        public void F1(int x)
        {
            this.x = x;
        }

        public static void F2(int x)
        {
            this.x = x; // Ошибка
        }
    }
```

В этом примере в записи «**this**.x = x» **this**.x – это обращение к члену класса, а x – к аргументу метода. Статические члены определены для класса, а не для экземпляра класса, поэтому для них данный идентификатор не определен.

Когда член объявлен как статический, имеется лишь одна его копия. Статический член создается при загрузке содержащего класс приложения и существует в течение жизни приложения. Поэтому мы можем обращаться к члену, даже если экземпляр класса еще не создан.

Один из примеров – метод Main. Среде CLR нужна универсальная точка входа в ваше приложение. Поскольку CLR не должна создавать экземпляры наших объектов автоматически (это прерогатива программиста), существует правило, требующее определить в одном из классов статический метод Main.

Некоторые члены класса обязаны быть статическими, в частности, перегруженные операторы (см. п. 4.7.3). Для других членов мы сами решаем – быть им статическими или нет. Если мы описываем член, формально принадлежащий классу, но не требующий реального объекта, то логично его сделать статическим. Например:

```
class MyClass1
{
      public int Sqr(int x)
      {
            return x * x;
      }
}
```

Здесь метод Sqr является методом экземпляра. Т.е. для его использования нужно создать экземпляр класса MyClass1:

```
MyClass1 cls1 = new MyClass1();
Console.WriteLine(cls1.Sqr(5));
```

Но зачем? Ведь данный метод не использует никакие другие члены экземпляра класса. Поэтому логичнее сделать так:

```
class MyClass2
{
      public static int Sqr(int x)
      {
            return x * x;
      }
}
```

Пользоваться таким методом проще:

```
Console.WriteLine(MyClass2.Sqr(5));
```

В следующем случае сделать методы статическими не получится:

```
class MyClass3
{
      public int x = 0;

      public int Sqr()
      {
            return x * x;
      }

      public static int Cube()
```

```
        {
            return x * x * x; // Ошибка
        }
    }
```

Здесь поле «x» является полем экземпляра, поэтому будет своим у каждого экземпляра класса. Ошибка связана с тем, что из одного же статического члена класса можно обращаться только к другим статическим членам. Из члена экземпляра можно получить доступ к любому члену класса.

Другой пример. Пусть нам нужно отслеживать число создаваемых экземпляров какого-либо класса. Это можно сделать следующим образом:

```
class MyClass4
{
    static public int Count = 0;

    public MyClass4()
    {
        Count++;
    }
}
```

Т.к. поле Count статическое, оно будет единым для всех классов. Даже если мы сами не знаем, сколько экземпляров класса MyClass4 было создано, в этом поле будет храниться точное значение:

```
MyClass4[] arr = new MyClass4[20];
Random rnd = new Random();

for (int i = 0; i < arr.Length; i++)
{
    if (rnd.Next(2) == 0) arr[i] = new MyClass4();
}

Console.WriteLine("Экземпляров: {0}", MyClass4.Count);
```

У статических полей класса должно быть некоторое допустимое значение. Его можно задать при определении (как это сделано в примере выше). Иначе ему будет присвоено значение по умолчанию.

Если класс объявлен как статический, то:

1) Он может содержать только статические члены. Константы (**const**) также считаются статическими полями.

2) Для него запрещено указывать список наследования. Единственное исключение – класс System.Object. Неявно все статические классы наследуются от него, и можно это указать явно.

3) От него нельзя наследоваться. По этой причине в статическом классе нельзя использовать модификаторы доступа **protected** и **protected internal**. –

они имеют смысл только при наличии классов-потомков.

4) Нельзя создавать его экземпляры. Т.к. все его члены являются статическими, то создавать, в принципе, нечего. Конструктор, если он объявлен, никогда не будет вызван.

Пример:

```
      static class MyStatClass1 { } // OK
      static class MyStatClass2 : MyStatClass1 { } // Ошибка

      static class MyStatClass3 : Object // OK
      {
            public int f1; // Ошибка
            public const int f2 = 1; // OK
            protected static int f3; // Ошибка
            internal static int f4 = 1; // OK

            static MyStatClass3()
            {
                  Console.WriteLine("Мы этого сообщения не увидим...");
            }

            public enum ZZZ { Z1, Z2, Z3 };
            public class ChildClass { }
      }

      class MyClass5 : MyStatClass1 {} // Ошибка

      static int Main()
      {
            MyStatClass3 sc1 = new MyStatClass3(); // Ошибка
            MyStatClass3.ChildClass sc2 = new MyStatClass3.ChildClass(); // OK

            Console.WriteLine("Мы используем MyStatClass3, но его конструктор
не вызывается");
            Console.WriteLine(MyStatClass3.f2);
            return 0;
      }
```

 Пример: Samples\4.2\4_2_3_static.

4.2.4. Создание и удаление экземпляров класса

Переменную, имеющую тип класса, можно инициализировать ссылкой на новый или уже существующий объект:

```
      class MyClass
      {
            public int x = 1;
      }

      static int Main()
      {
            MyClass cls1 = new MyClass();
            MyClass cls2 = cls1;
```

```
            cls1.x = 5;
            Console.WriteLine(cls2.x); // 5
            return 0;
      }
```

Если мы хотим, чтобы экземпляр cls2 был копией экземпляра cls1, а не ссылкой на него, нужно использовать структуры. Копирование же классов не предусмотрено. Хотя можно самостоятельно описать метод, копирующий класс:

```
      class MyClass
      {
            public int x = 1;

            public MyClass Copy()
            {
                  MyClass cls = new MyClass();
                  cls.x = x;
                  return cls;
            }
      }

      static int Main()
      {
            MyClass cls1 = new MyClass();
            MyClass cls3 = cls1.Copy();

            cls1.x = 5;
            Console.WriteLine(cls3.x); // 1
            return 0;
      }
```

Чтобы как-то систематизировать копирование классов (ведь методы копирования каждый разработчик может сделать по-своему), используется интерфейс ICloneable (см. п. 4.9.4.3).

Запрещено создавать экземпляры статических и абстрактных классов. Структуры, как мы уже отмечали, могут быть созданы и без использования оператора **new** (см. п. 3.1.2.6).

При описании операторов языка C# мы ничего не говорили об операторе удаления объектов **delete**. Дело в том, что такого оператора в языке C#, в отличие от языка C++, нет. Все объекты в управляемой динамической куче, на которые не осталось ссылок, удаляются сборщиком мусора автоматически.

 Пример: Samples\4.2\4_2_4_constr.

4.2.5. Вложенные типы

В классе можно описывать новые типы данных – *вложенные типы*. Полным именем вложенного типа будет

```
<имя класса>.<имя вложенного типа>
```

Это могут быть:

• перечисления (п. 3.1.2.5);

• структуры (п. 3.1.2.6);

• классы;

• делегаты (§ 4.8);

• интерфейсы (§ 4.9).

Если в классе объявлен новый тип данных, то его можно использовать при описании других членов класса (как тип полей, тип аргументов метода и т.п.). Однако, при этом уровень доступа к такому члену должен быть не менее строгим, чем уровень доступа к вложенному типу. Иначе компилятор диагностирует ошибку «Несовместимость по доступности».

Пример:

```
class MyClass
{
      protected enum MyEnum { };
      public MyEnum field1; // Ошибка
      private MyEnum field2; // ОК

      private struct MyStruct { }
      public void F1(MyStruct st) { } // Ошибка
      private void F2(MyStruct st) { } // ОК
}
```

 Пример: Samples\4.2\4_2_5_types.

§ 4.3. Описание полей класса

Здесь и далее при описании синтаксиса членов класса мы не будем упоминать атрибуты, с ними разберемся в § 5.4. Также не будем говорить о модификаторах, за исключением тех случаев, когда они оказывают существенное влияние на поведение члена класса. Доступные же для различных членов класса модификаторы перечислены выше.

4.3.1. Константы

Синтаксис:

```
<константа> :: [<атрибуты>] [<модификаторы>] const <оператор объявления>
```

Тип, указанный в объявлении константы, должен быть типом **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, **double**, **decimal**, **bool**, **string**, перечисляемым или ссылочным типом. Таким образом, например, константа не может иметь тип структуры. Константа отличного от **string** ссылочного типа может инициализироваться только значением **null**.

Оператор объявления (см. п. 3.4.1) обязательно должен быть с инициализацией, причем инициализирующее выражение должно быть константным. Константы могут использоваться в константных выражениях, только если они не являются циклическими.

Пример:

```
int a = 1;
const int x = 1; // OK
const int y = x * x + x; // OK
const int b = a + 1; // Ошибка

struct S { }
const S q = new S { }; // Ошибка

const int z = z + 1; // Ошибка
const int a2 = b2 + 1; // Ошибка
const int b2 = a2 + 1; // Ошибка

const string S1 = "!!!"; // OK
const Type S2 = typeof(int); // Ошибка
```

 Пример: Samples\4.3\4_3_1_const.

4.3.2. Поля

Синтаксис:

```
<поле> :: [<атрибуты>] [<модификаторы>] <тип> <оператор объявления>
```

Как видно из таблицы 4.2, среди прочих модификаторов для полей применимы модификаторы **static** и **readonly**, причем их можно комбинировать. Чем будут отличаться поля, описанные с такими модификаторами и без них? И чем они отличаются от констант? Отличия приведены в табл. 4.3.

Табл. 4.3 – Отличия модификаторов полей класса

	const	–	**readonly**	**static**	**static readonly**
Инициализация при объявлении	нужно	можно	можно	можно	можно
Можно инициализировать в статическом конструкторе	–	–	–	+	+
Можно инициализировать в конструкторе экземпляра	–	+	+	+	–
Можно инициализировать в статическом методе	–	–	–	+	–
Можно инициализировать в методе экземпляра	–	+	–	+	–
Модификация извне класса	–	+	–	+	–
Инициализация значением, которое невозможно вычислить во время компиляции	–	+	+	+	+

И статические и экземплярные переменные, если их не инициализировать при объявлении в классе, автоматически инициализируются значением по умолчанию. Инициализация полей структуры при их объявлении, как уже было сказано, запрещена.

Пример:

```
class MyClass
{
```

```
        const int Field10; // Ошибка
        public int Field11; // Предупреждение
        public readonly int Field12; // Предупреждение
        public static int Field13; // Предупреждение
        public static readonly int Field14; // Предупреждение

        public const int Field20 = 0; // OK
        public int Field21 = 1; // OK
        public readonly int Field22 = 2; // OK
        public static int Field23 = 3; // OK
        public static readonly int Field24 = 4; // OK

        public const double Field30 = Math.Sqrt(2.0); // Ошибка
        public double Field31 = Math.Sqrt(2.0); // OK
        public readonly double Field32 = Math.Sqrt(2.0); // OK
        public static double Field33 = Math.Sqrt(2.0); // OK
        public static readonly double Field34 = Math.Sqrt(2.0); // OK

        static MyClass()
        {
              Field20 = 0; // Ошибка
              Field21 = 1; // Ошибка
              Field22 = 2; // Ошибка
              Field23 = 3; // OK
              Field24 = 4; // OK
        }

        public MyClass()
        {
              Field20 = 0; // Ошибка
              Field21 = 1; // OK
              Field22 = 2; // OK
              Field23 = 3; // OK
              Field24 = 4; // Ошибка
        }

        public static void Method2()
        {
              Field20 = 0; // Ошибка
              Field21 = 1; // Ошибка
              Field22 = 2; // Ошибка
              Field23 = 3; // OK
              Field24 = 4; // Ошибка
        }

        public void Method1()
        {
              Field20 = 0; // Ошибка
              Field21 = 1; // OK
              Field22 = 2; // Ошибка
              Field23 = 3; // OK
              Field24 = 4; // Ошибка
        }
  }

  static int Main()
  {
        MyClass cls = new MyClass();
        MyClass.Field20 = 1; // Ошибка
        cls.Field21 = 1; // OK
```

```
            cls.Field22 = 2; // Ошибка
            MyClass.Field23 = 3; // ОК
            MyClass.Field24 = 4; // Ошибка
            return 0;
        }
```

 Пример: Samples\4.3\4_3_2_fields.

§ 4.4. Описание методов класса

Среди методов классов можно выделить:

- методы экземпляров;
- статические методы;
- конструкторы экземпляров;
- статические конструкторы;
- деструкторы;
- метод Main.

4.4.1. Синтаксис описания методов

Общий синтаксис описания методов следующий:

```
<метод> :: [<атрибуты>] [<модификаторы>] [static]
<тип возвращаемого значения> <имя метода> [<параметры типа>]
([<список формальных параметров>]) [<ограничения параметров типа>]
<тело метода>

<имя метода> :: <идентификатор>
<имя метода> :: <тип интерфейса>.<идентификатор>

<тело метода> :: <блок>
<тело метода> :: ;

<список формальных параметров> :: <фиксированный параметр> [, ...]

<фиксированный параметр> :: [<атрибуты>] [<модификатор параметра>]
      <тип> <идентификатор>

<модификатор параметра> :: ref
<модификатор параметра> :: out
<модификатор параметра> :: this
<модификатор параметра> :: params
```

Как уже было сказано выше, параметры типа будут рассмотрены в § 5.1. Типом возвращаемого значения может быть любой рассмотренный нами ранее тип данных, а также специальный тип **void** (псевдоним структуры System.Void), который означает, что метод ничего не возвращает.

Синтаксис вызова статического метода и метода экземпляра будет следующим:

```
      <вызов статического метода> :: <имя класса>.<идентификатор> ([<список
фактических параметров>])
      <вызов метода экземпляра> :: <экземпляр класса>.<идентификатор>
([<список фактических параметров>])

      <список фактических параметров> :: <фактический параметр> [, ...]
```

```
<фактический параметр> :: <выражение>
<фактический параметр> :: ref <l-value>
<фактический параметр> :: out <l-value>
```

Тип фактического параметра должен совпадать с типом соответствующего формального параметра. Также должны быть указаны модификаторы **ref** или **out**, если они были указаны у формального параметра.

Два разных метода могут иметь одинаковые имена, если:

1. Они различаются списком формальных аргументов;
2. Один из методов явно реализует метод интерфейса.

Пример:

```
interface IMyInterface
{
	void Method();
}

partial class MyClass : IMyInterface
{
	public void Method()
	{
		Console.WriteLine("Method");
	}

	void IMyInterface.Method()
	{
		Console.WriteLine("IMyInterface.Method");
	}

	private void Method(int x) { }
	private void Method() { } // Ошибка

	partial void Part();

	partial void Part()
	{
		Console.WriteLine("Part");
	}
}

class Program
{
	static int Main()
	{
		MyClass cls = new MyClass();
		cls.Method(); // Method
		((IMyInterface)cls).Method(); // IMyInterface.Method
		return 0;
	}
}
```

Тело у метода может отсутствовать, если он является абстрактным, внешним или разделяемым. Разделяемый метод не может возвращать значе-

ние и не имеет модификаторов доступа (т.е. всегда является **private**).

 Пример: Samples\4.4\4_4_1_methods.

4.4.1.1. Виды формальных параметров

В языке C++ предусмотрена передача параметров по значению, по ссылке и по указателю. В первом случае в методе создается локальная копия параметра, и все изменения локального параметра никак не отражаются на исходном значении:

```
void f1(int i)
{
	i = 5;
}

void main(void)
{
	int x = 1;
	f(x); // x = 1
}
```

В результате переменная «x» сохранит свое значение 1. При передаче параметра по ссылке или по указателю в метод передается адрес переменной, поэтому при изменении локального параметра меняется и исходное значение:

```
void f2(int &i)
{
	i = 5;
}

void f3(int *i)
{
	*i = 5;
}

void main(void)
{
	int y = 1;
	int z = 1;
	f2(y); // y = 5
	f3(&z); // z = 5
}
```

При этом требуется, чтобы у параметра можно было вычислить адрес (т.е. чтобы оно являлось l-value). Например, следующие вызовы приведут к ошибке компиляции:

```
f2(5); // Ошибка
f2(z + y); // Ошибка
```

В языке C# существует схожее решение. Без указания модификаторов

формальный параметр считается *входным*. В метод передается его копия. Чтобы сделать параметр *выходным* (т.е. чтобы через него можно было вернуть выходные результаты работы метода), его нужно объявить с модификатором **ref** или **out**. В этом случае передается ссылка на параметр. Разница между этими модификаторов в том, что **ref** требует обязательной инициализации параметра перед использованием. А модификатор **out** требует обязательной инициализации параметра в вызываемом методе до передачи управления в вызывающий метод. Выходные параметры, как и в языке C++, обязательно должны являться l-value:

```
static void MetI(int i) { i = 5; }
static void MetR(ref int i) { i = 7; }
static void MetO(out int i) { i = 9; }
static void MetWAR(ref int i) { } // ОК
static void MetWAO(out int i) { } // Ошибка

static int Main()
{
      int i = 1;
      int j;

      MetI(i);
      Console.WriteLine(i); // 1
      MetR(ref i);
      Console.WriteLine(i); // 7
      MetO(out i);
      Console.WriteLine(i); // 9

      MetI(j); // Ошибка
      MetR(ref j); // Ошибка
      MetO(out j); // ОК

      MetO(out 5); // Ошибка
      MetO(out i + j); // Ошибка
      return 0;
}
```

Считается, что сигнатура параметра с модификатором **ref** или **out** отличается от сигнатуры параметра без модификаторов, но сигнатура **ref** не отличается от сигнатуры **out**:

```
void F(int i) { }
void F(ref int i) { } // ОК
void F(out int i) { } // Ошибка
```

4.4.1.2. Методы расширения

Модификатор **this** указывает, что описываемый метод является *методом расширения*. Он может быть указан только у первого параметра. Методы расширения могут быть объявлены только в статических классах, не являю-

щихся универсальными или вложенными. Комбинировать модификатор **this** с другими модификаторами нельзя. При следующем объявлении метода расширения

```
      static <тип результата> <имя метода> (this <тип> <идентификатор> [,
<список формальных параметров>])
```

в результате мы расширяем тип <тип> новым методом экземпляра

```
      <тип результата> <тип>.<имя метода> ([<список формальных параметров>])
```

Пример:

```
      static class ExtMethod
      {
            public static string X(this string s)
            {
                  return "ExtMethod.X";
            }
      }

      class Program
      {
            static int Main()
            {
                  string s = String.Empty;

                  Console.WriteLine(s.X()); // ExtMethod.X
                  return 0;
            }
      }
```

В данном примере мы в класс String добавили новый метод X().

4.4.1.3. Переменное число параметров метода

В языке C++ существует возможность описывать методы с переменным числом параметров, используя многоточие, например:

```
      void f(int count, ...);
```

Были предусмотрены специальные средства для работы с такими параметрами. В языке C# все проще: т.к. все типы данных имеют единый корень System.Object, любое количество параметров можно представить в виде массива **object**[] или массива производных от Object классов – *массива параметров*. Как и в языке C++, в списке параметров может быть только один массив параметров, и он должен быть последним в списке. Для описания массива параметров используется модификатор **params**:

```
      static void Write() { Console.WriteLine("()"); }
      static void Write(int i, int j) { Console.WriteLine("(int,int)"); }
      static void Write(params int[] i) { Console.WriteLine("(int[])"); }
```

```
static int Main()
{
      Write();         // ()
      Write(1);        // (int[])
      Write(1, 1);     // (int,int)
      Write(1, 1, 1); // (int[])
      return 0;
}
```

Видно, что компилятор сам подбирает метод с наиболее подходящей сигнатурой. Комбинировать **params** с другими модификаторами нельзя. Тип данных в массиве параметров может быть любой, но он обязательно должен быть массивом, причем прямоугольные массивы использовать нельзя:

```
static void Write(params object[][] obj) { } // ОК
static void Write(params string[,] str) { } // Ошибка
```

4.4.1.4. Передача структуры и ссылки класса в метод

Все, сказанное про входные и выходные параметры, верно для типов данных по значению. А что насчет классов? Ведь они и так являются ссылочными типами, следовательно, в метод будет в любом случае передаваться ссылка на экземпляр класса:

```
static void Met(MyClass2 cls) { cls.a = 5; }
static void Met(ref MyClass2 cls) { cls.a = 7; }
static void Met(MyStruct2 str) { str.a = 5; }
static void Met(ref MyStruct2 str) { str.a = 7; }

static int Main()
{
      MyClass2 cls2 = new MyClass2();
      MyStruct2 str2 = new MyStruct2();

      Met(cls2);
      Met(str2);
      Console.WriteLine(cls2.a); // 5
      Console.WriteLine(str2.a); // 0
      Met(ref cls2);
      Met(ref str2);
      Console.WriteLine(cls2.a); // 7
      Console.WriteLine(str2.a); // 7
      return 0;
}
```

Видим, что в данном случае для класса нет никакой разницы, указан модификатор **ref** или нет, а поведение структуры отличается. На самом деле, разница есть, но только если модифицировать не содержимое класса, а ссылку на класс. Т.е. ссылка тоже может быть передана как входной и как выходной параметр:

```
static void MetR(MyClass2 cls) { cls = null; }
static void MetR(ref MyClass2 cls) { cls = null; }

static int Main()
{
      MyClass2 cls2 = new MyClass2();
      MetR(cls2);
      Console.WriteLine(cls2 == null); // False
      MetR(ref cls2);
      Console.WriteLine(cls2 == null); // True
      return 0;
}
```

Посмотрим на рис. 4.6. В первом случае в параметр cls в методе MetR копировалась ссылка на экземпляр класса, лежащий в динамической куче. Поэтому мы, меняя поле класса cls.a, меняли тем самым и поле класса cls2.a. Но, обнулив ссылку cls, мы не повлияли на cls2. Во втором же случае cls неявно является ссылкой на cls2. Поэтому, когда мы пишем cls = **null**, тем самым мы обнуляем ссылку cls2.

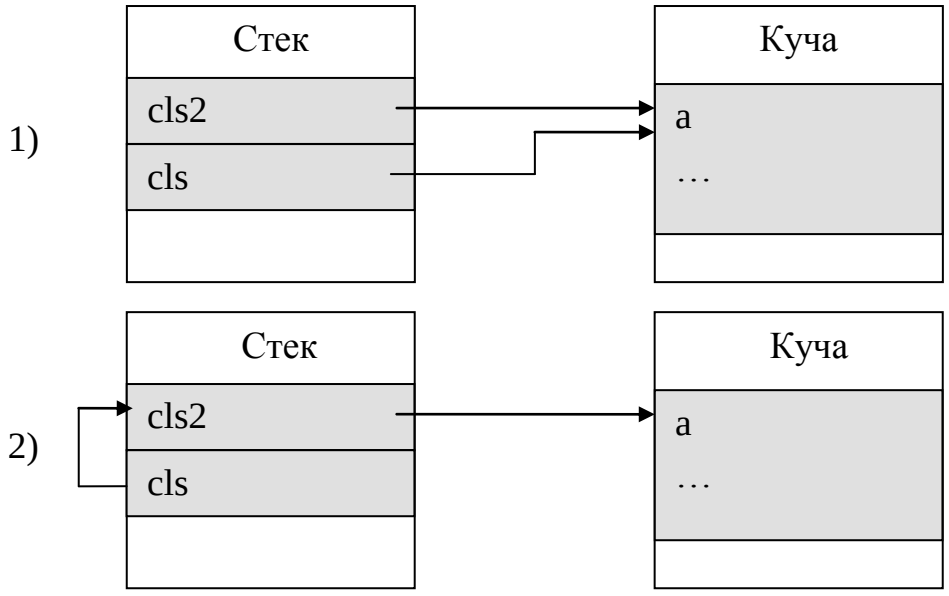


Рис. 4.6 – Передача ссылки по значению и по ссылке

В языке C++ тоже существует такая возможность – указатели в методы можно передавать не только по значению, но также по ссылке и по указателю.

4.4.2. Конструкторы

Конструкторы – это специальные методы, вызывающиеся в момент создания экземпляра класса. Экземпляры класса создаются при помощи оператора **new**, всегда имеющего определенный тип возвращаемого значения:

```
<экземпляр класса> operator new <класс>([<параметры конструктора>])
```

Поэтому в описании конструктора не должно быть указано никакого возвращаемого значения, даже **void**.

В языке C++ существует две возможности создать экземпляр класса – на стеке и в динамической куче:

```
MyClass cls1; // на стеке без параметров
MyClass cls2(<параметры>); // на стеке с параметрами
MyClass cls3 = cls1; // на стеке копированием
MyClass *cls4 = new MyClass; // в куче без параметров
MyClass *cls5 = new MyClass(<параметры>); // в куче с параметрами
```

Причем конструкторы в языке C++ делятся на три группы, имеющие существенные особенности – конструкторы по умолчанию, конструкторы копирования и все прочие конструкторы с параметрами. Из-за того, что в языке C# копирование объектов было серьезно переосмыслено, конструкторы копирования потеряли свою значимость. Вариант №3 теперь просто копирует ссылку, а остальные варианты вызовут ошибку компиляции. Поэтому в языке C# различают три вида конструкторов:

• конструкторы по умолчанию,

• остальные конструкторы экземпляров,

• и статические конструкторы.

Хотя ссылка является, по сути, указателем, это скрыто от программиста. Нет никакой возможности (в управляемом коде) получить реальный адрес объекта. Поэтому, в отличие от синтаксиса C++, символ указателя опускается, хотя выделение памяти происходит похожим образом:

```
MyClass cls1 = new MyClass(); // в куче без параметров
MyClass cls2 = new MyClass(<параметры>); // в куче с параметрами
MyClass cls3 = cls1; // копирование ссылки
```

Как и другие методы, конструктор может быть полиморфным, т.е. иметь несколько реализаций, различающихся формальными аргументами. Синтаксис:

```
<конструктор> :: [<атрибуты>] [<модификаторы>] [static] <имя класса>
      ([<список формальных параметров>]) [<инициализатор>]
<тело конструктора>

<тело конструктора> :: <блок>
<тело конструктора> :: ;

<инициализатор> :: : base ([<список фактических параметров>])
<инициализатор> :: : this ([<список фактических параметров>])
```

Как и в языке C++, имя конструктора совпадает с именем класса. Тело

может отсутствовать у внешнего (**extern**) конструктора.

 Пример: Samples\4.4\4_4_2_constr.

4.4.2.1. Конструкторы по умолчанию

Если в классе не объявлен ни один конструктор, то компилятором автоматически добавляется конструктор по умолчанию. Его название следует из его функции – инициализации полей класса значениями по умолчанию. Если в классе конструктор по умолчанию не описан, но есть конструкторы с параметрами, то использование конструктора по умолчанию запрещено:

```
class MyClass1
{
}

class MyClass2
{
      public MyClass2() { Console.WriteLine("MyClass2()"); }
      public MyClass2(int i) { Console.WriteLine("MyClass2(int)"); }
}

class MyClass3
{
      public MyClass3(int i) { Console.WriteLine("MyClass3(int)"); }
}

struct MyStruct
{
      public MyStruct() { } // Ошибка
      public MyStruct(int i) { Console.WriteLine("MyStruct(int)"); }
}

static int Main()
{
      MyClass1 cls1 = new MyClass1();
      MyClass2 cls21 = new MyClass2();
      MyClass2 cls22 = new MyClass2(1);
      MyClass3 cls31 = new MyClass3(); // Ошибка
      MyClass3 cls32 = new MyClass3(1);
      MyStruct str1 = new MyStruct(); // OK
      MyStruct str2 = new MyStruct(1);

      return 0;
}
```

В структуре нельзя описать явно конструктор по умолчанию, но он всегда определен в ней неявно.

4.4.2.2. Инициализаторы конструкторов

Перед тем, как выполнится конструктор экземпляра, можно выполнить

его инициализатор. Инициализаторы бывают двух видов:

• Вида **base**(...), который активизирует конструктор экземпляра базового класса (см. п. 4.6.2);

• Вида **this**(...), который позволяет текущему классу вызвать другой конструктор экземпляра, определенный в нем самом. В скобках указываются фактические параметры вызываемого конструктора.

Это полезно, когда, например, конструктор по умолчанию содержит некий общий код инициализации объекта. Тогда конструктор с параметрами может сначала вызвать конструктор по умолчанию для совершения этих действий, а затем уже выполнит свой код.

Пример:

```
      class MyClass4
      {
            public MyClass4() { Console.WriteLine("MyClass4()"); }
            public MyClass4(int i) : this() {
Console.WriteLine("MyClass4(int)"); }
            public MyClass4(int i, int j) : this(1) {
Console.WriteLine("MyClass4(int,int)"); }
            public MyClass4(char c) : this(c) { } // Ошибка
      }
```

4.4.2.3. Статические конструкторы

Конструкторы также могут быть статическими. Они предназначены для инициализации статических полей класса, если это требуется. Если конструктор статический, то он один для всех экземпляров класса. Явно его вызывать нельзя, CLR делает это автоматически. Причем предсказать момент вызова статического конструктора невозможно. Однако гарантируется, что это произойдет до момента использования класса (вызова его статического метода, создания экземпляра и т.п.). Пример:

```
      class MyClass5
      {
            static MyClass5() { Console.WriteLine("static MyClass5"); }
            public MyClass5() { Console.WriteLine("public MyClass5"); }
      }

      class MyClass6
      {
            static MyClass6(int q) { } // Ошибка
            static MyClass6() { Console.WriteLine("static MyClass6"); }
      }

      static int Main()
      {
            MyClass5 cls5 = new MyClass5(); // OK
```

```
            MyClass6 cls6 = new MyClass6(); // OK

            return 0;
        }
```

В данном случае строка «static MyClass» всегда будет появляться на консоли раньше, чем «public MyClass». Сигнатура статического конструктора может совпадать с сигнатурой конструктора по умолчанию, это не считается ошибкой. Соответственно, если в классе нет других конструкторов, кроме статического, то неявно добавляется конструктор по умолчанию.

Статические конструкторы не могут иметь модификаторов доступа и параметров. Статический конструктор статического класса никогда не будет вызван.

4.4.3. Деструкторы

Деструктор – это метод, который вызывается при уничтожении объекта. Так как в .NET CLR используется сборщик мусора, нельзя явно удалить определенный экземпляр. Удаление происходит, когда никакой код уже не может воспользоваться этим экземпляром (т.е. когда не остается ссылок на данный экземпляр).

Синтаксис:

```
<деструктор> :: [<атрибуты>] ~<имя класса> () <тело деструктора>

<тело деструктора> :: <блок>
<тело деструктора> :: ;
```

Имя деструктора состоит из тильды и имени класса. Параметры отсутствуют, модификаторы доступа не указываются. Как уже было сказано в пункте 4.2.2, для класса, содержащего деструктор, зарезервирована сигнатура «**void** Finalize()». Если деструктор внешний (**extern**), то его тело в классе отсутствует.

Статические классы и структуры не могут содержать деструкторы.

4.4.4. Метод Main

У каждого приложения на C# должен быть метод Main, определенный в одном из его классов. Кроме того, этот метод должен быть определен как **public** (хотя в последних версиях .NET Framework это не обязательно) и **static**. Для компилятора C# не важно, в каком из классов определен метод Main, а класс, выбранный для этого, не влияет на порядок компиляции.

В C# разработчиками включен механизм, позволяющий определять более одного класса с методом Main. Зачем это нужно? Одна из причин – необходимость поместить в наши классы тестовый код. Затем, используя переключатель

```
csc.exe /main:<имя класса> ...
```

компилятора C#, можно задавать класс, метод Main которого должен быть задействован.

Также можно выбрать этот класс в опциях проекта в среде Visual Studio 2008 (меню «Проект» → «Свойства…», закладка «Приложение», выпадающий список «Автоматически запускаемый объект»).

§ 4.5. Свойства. Индексаторы

Свойство – это член, предоставляющий доступ к характеристикам объекта или класса. Свойства – это естественное расширение полей. Как свойства, так и поля являются именованными членами со связанными типами, для обращения к которым используется одинаковый синтаксис. Однако, в отличие от полей, свойства не указывают места хранения. Вместо этого свойства содержат методы доступа, определяющие операторы, которые используются при чтении или записи их значений. Таким образом, свойства предоставляют механизмы, позволяющие связать определенные действия с чтением или записью атрибутов объекта. Кроме того, свойства обеспечивают вычисление таких атрибутов.

4.5.1. Определение и использование свойств

Всегда поощряется создание классов, которые не только скрывают реализацию своих методов, но и запрещают любой прямой доступ к полям класса. Обеспечить корректную работу с полем можно, предоставляя методы доступа (accessor methods), выполняющие работу по получению и установке значений этих полей, так чтобы они действовали согласно правилам конкретной предметной области. Такая конструкция называется *свойством* класса.

Например, в языке C++ это можно сделать так:

```
class MyClass
{
private:
      int F;
public:
      int GetF(void) const { /* стратегия чтения */ }
      void SetF(int f) { /* стратегия записи */ }
};
```

В расширениях языка C++, появившихся в библиотеке Borland VCL, предусмотрены специальные языковые конструкции для описания свойств:

```
class MyClass
{
private:
      int FF;
      int __fastcall GetF(void) { /* стратегия чтения */ }
      void __fastcall SetF(int f) { /* стратегия записи */ }
public:
      __property int F = {read = GetF, write = SetF};
```

```
};
```

Так гораздо удобнее – со свойством можно работать как с обычным полем, не задумываясь о том, что при доступе к нему вызываются какие-то скрытые методы. Эту идею заимствовали и разработчики языка C#, хотя несколько изменили синтаксис.

Свойство в C# состоит из объявления поля и методов доступа, применяемых для изменения значения поля. Эти методы доступа называются *получатель* (getter) и *установщик* (setter). Методы-получатели используются для получения значения поля, а установщики – для его изменения. Общий синтаксис таков:

```
<свойство> :: [<атрибуты>] [<модификаторы>] <тип> <имя свойства>
"{" <методы доступа> "}"

<имя свойства> :: <идентификатор>
<имя свойства> :: <тип интерфейса>.<идентификатор>

<методы доступа> :: <получатель> [<установщик>]
<методы доступа> :: <установщик> [<получатель>]

<получатель> :: [<атрибуты>] [<модификатор метода>] get <тело метода>
<установщик> :: [<атрибуты>] [<модификатор метода>] set <тело метода>

<модификатор метода> :: internal
<модификатор метода> :: protected
<модификатор метода> :: protected internal
<модификатор метода> :: internal protected
<модификатор метода> :: private

<тело метода> :: <блок>
<тело метода> :: ;
```

Как видно, у получателя и установщика могут быть разные уровни доступа. Тело у свойства может отсутствовать, если оно абстрактное, внешнее или автоматически реализуемое. Наследование и перегрузка свойств обсуждаются в § 4.7.

 Пример: Samples\4.5\4_5_1_props.

4.5.1.1. Методы доступа

Метод доступа **get** (получатель) соответствует не содержащему параметров методу, возвращаемое значение которого имеет тип свойства. Все операторы **return** в теле получателя должны задавать выражение, которое может быть неявно преобразовано к типу свойства. Кроме того, конечная точка получателя должна быть недостижима.

Метод доступа **set** (установщик) соответствует методу с типом возвращаемого значения **void** и одним параметром значения, имеющим тип свойства. Неявному параметру установщика всегда присваивается имя **value.** Операторы **return** в теле установщика не могут задавать выражение.

В зависимости от наличия или отсутствия получателя и установщика свойства классифицируются следующим образом:

• Свойство, содержащее оба метода доступа, называется *свойством для чтения и записи*.

• Свойство, содержащее только получатель, называется *свойством только для чтения*. При использовании свойства только для чтения в качестве конечного объекта для операции присваивания возникает ошибка времени компиляции.

• Свойство, содержащее только установщик, называется *свойством только для записи*. Свойство только для записи может использоваться в выражениях только в качестве конечного объекта операции присваивания. Во всех остальных случаях возникает ошибка времени компиляции.

Пример:

```
public class Person1
{
      public enum TSex { Male, Female };

      private TSex FSex;
      private string FName;
      private int FAge = 0;

      public string Name
      {
            set { FName = value; }
            get { return FName; }
      }

      public TSex Sex
      {
            protected set { FSex = value; }
            get { return FSex; }
      }

      public int Age
      {
            set
            {
                  if (value >= 0)
                  {
                        FAge = value;
                  }
                  else
                  {
```

```
                              throw new ArgumentOutOfRangeException("Возраст
не должен быть отрицательным");
                        }
                  }
                  get
                  {
                        return FAge;
                  }
            }

            public Person1(TSex sex)
            {
                  FSex = sex;
                  FName = sex == TSex.Male ? "Иван" : "Маша";
            }
      }

      static int Main()
      {
            Person1 p1 = new Person1(Person1.TSex.Male);

            p1.Name = "Пётр";
            p1.Age = -1; // ArgumentOutOfRangeException
            p1.Age = 7;
            p1.Sex = Person1.TSex.Female; // Ошибка
            return 0;
      }
```

В классе Person объявлены три свойства. Свойство Name не содержит ограничений. Свойство Age генерирует исключительную ситуацию, если попытаться установить отрицательный возраст. Свойство Sex задается только один раз, в конструкторе, а потом извне класса его можно только читать.

Итак, свойства позволяют предоставлять методы доступа для полей и универсальные, интуитивно понятные интерфейсы для клиента. Из-за этого свойства иногда называют «умными полями».

4.5.1.2. Автоматически реализуемые свойства

Если для обоих методов доступа мы не указываем тело, то получаем *автоматически реализуемое свойство*. Для него автоматически создается скрытое резервное поле, для которого реализуются методы доступа для чтения и записи.

Поскольку резервное поле недоступно, чтение и запись его значений осуществляются только с помощью методов доступа свойства. Это означает, что автоматически реализуемые свойства только для чтения или только для записи не имеют смысла и не разрешены. Однако возможно задать различные уровни доступа для каждого из методов доступа.

Например, сделаем поле Sex в предыдущем примере автоматически ре-

ализуемым:

```
      public class Person2
      {
            public enum TSex { Male, Female };

            private string FName;
            private int FAge = 0;

            public string Name
            {
                  set { FName = value; }
                  get { return FName; }
            }

            public TSex Sex
            {
                  protected set;
                  get;
            }

            public int Age
            {
                  set
                  {
                        if (value >= 0)
                        {
                              FAge = value;
                        }
                        else
                        {
                              throw new ArgumentOutOfRangeException("Возраст
не должен быть отрицательным");
                        }
                  }
                  get
                  {
                        return FAge;
                  }
            }

            public Person2(TSex sex)
            {
                  Sex = sex;
                  FName = sex == TSex.Male ? "Иван" : "Маша";
            }
      }

      static int Main()
      {
            Person2 p2 = new Person2(Person2.TSex.Female);

            p2.Age = 12;
            return 0;
      }
```

4.5.1.3. Доступность

На использование модификаторов методов доступа налагаются следующие ограничения:

• Модификатор метода доступа не может использоваться в интерфейсе или явной реализации члена интерфейса;

• Для свойства или индексатора, не имеющего модификатора **override**, модификатор метода доступа может использоваться только в том случае, если свойство или индексатор содержит оба метода доступа, и применяется только к одному из них;

• Для свойства или индексатора, содержащего модификатор **override**, метод доступа должен соответствовать используемому модификатору метода доступа переопределяемого метода доступа;

• Модификатор метода доступа должен объявлять более строгий уровень доступа, чем уровень доступа самого свойства или индексатора.

После выбора конкретного свойства или индексатора допустимость их использования определяется на основании областей доступности задействованных методов доступа:

• Для использования в качестве значения (r-value) должен существовать и быть доступным получатель.

• Для использования в качестве конечного объекта операции простого присваивания должен существовать и быть доступным установщик.

• Для использования в качестве конечного объекта операции составного присваивания или в качестве конечного объекта операторов инкремента и декремента должны существовать и быть доступны оба метода доступа.

Как уже было сказано в пункте 4.2.2, для свойства P типа T зарезервированы сигнатуры «T get_P()» и «**void** set_P(T value)» (независимо от типа свойства). Методы с такими сигнатурами для доступа к свойству создаются компилятором автоматически.

Пример:

```
public class Test
{
      int x = 0, y = 0, z = 0;
      public int X { get { return x;  } }
      public int Y { set { y = value; } }
      public int Z { get { return z; } set { z = value; } }
}

public class Props
```

```
{
      int X1 { get; set; } // Ошибка
      void get_X1() { }

      int X2 { get; set; } // OK
      void get_X2(int i) { }
}

static int Main()
{
      Test test = new Test();
      int val;

      test.X = val; // Ошибка
      test.X++; // Ошибка
      val = test.X; // OK
      test.Y = val; // OK
      val = test.Y; // Ошибка
      test.Y++; // Ошибка
      val = test.Z; // OK
      test.Z = val; // OK
      test.Z++; // OK
      return 0;
}
```

4.5.2. Индексаторы

Прейдем к рассмотрению *индексаторов* – особой возможности C#, позволяющей программно обращаться с объектами так, как если бы они были массивами. Индексаторы примерно соответствуют возможности перегрузки операции индексации в языке C++:

```
class MyClass
{
public:
      int &operator [] (int idx) { /* стратегия доступа */ }
}
```

В этом примере оператор индексации возвращает ссылку, чтобы его можно было использовать и для чтения, и для записи. В расширениях языка C++, введенных в библиотеке Borland VCL, появилась возможность делать индексируемые свойства. Причем тип и количество индексов можно варьировать (тогда как в операторе индексации это всегда одно целое число):

```
class MyClass
{
private:
      int __fastcall GetF(String idx) { /* стратегия чтения */ }
      void __fastcall SetF(String idx, int f) { /* стратегия записи */ }
public:
      __property int F[String] = {read = GetF, write = SetF};
};
```

В данном примере в качестве индекса выступает строка. По этому же пути пошли разработчики языка C#.

Синтаксис описания индексаторов совпадает с таковым для свойств, за исключением имени:

```
      <имя свойства> :: this "[" <список формальных параметров > "]"
      <имя свойства> :: <тип интерфейса>.this "[" <список формальных
параметров > "]"
```

Индексаторов, отличающихся списками формальных аргументов, в классе может быть несколько. Использование модификаторов **ref** и **out** запрещено. При работе с индексатором класс ведет себя как массив:

```
      <доступ к индексатору> :: <экземпляр класса> "[" <список фактических
параметров > "]"
```

Например, добавим в разработанный нами ранее класс Person информацию о весе субъекта на каждом году жизни:

```
      public class Person3
      {
            public enum TSex { Male, Female };

            private string FName;
            private int FAge = 0;
            double[] FWeights;

            public string Name
            {
                  set { FName = value; }
                  get { return FName; }
            }

            public TSex Sex
            {
                  protected set;
                  get;
            }

            public int Age
            {
                  set
                  {
                        if (value >= 0)
                        {
                              FAge = value;
                              FWeights = new double[FAge + 1];
                        }
                        else
                        {
                              throw new ArgumentOutOfRangeException("Возраст
не должен быть отрицательным");
                        }
                  }
                  get
```

```
                {
                        return FAge;
                }
        }

        public double this[int year]
        {
                get { return FWeights[year]; }
                set { FWeights[year] = value; }
        }

        public Person3(TSex sex)
        {
                Sex = sex;
                FName = sex == TSex.Male ? "Иван" : "Маша";
                FWeights = new double[1] { 3.5 };
        }
    }

    static int Main()
    {
        Person3 p3 = new Person3(Person3.TSex.Male);

        p3.Name = "Саша";
        p3.Age = 3;
        p3[0] = 3.2;
        p3[1] = 5.0;
        p3[2] = 7.4;
        p3[3] = 9.8;

        for(int i = 0; i <= p3.Age; i++)
        {
                Console.WriteLine("Когда {0} был(а) в возрасте {1} лет,
его(её) вес составлял {2} кг.", p3.Name, i, p3[i]);
        }

        return 0;
    }
```

Как уже было сказано в пункте 4.2.2, для индексатора типа T со списком формальных параметров L зарезервированы сигнатуры «T get_Item(L)» и «**void** set_Item(L, T value)». Пример:

```
    class Indexers
    {
        int this[int i] { get { return 0; } } // Ошибка
        int this[int i, int j] { get { return 0; } } // ОК
        int get_Item(int x) { return 0; }
    }
```

 Пример: Samples\4.5\4_5_2_index.

§ 4.6. Наследование

Чтобы построить – в терминах данных или поведения – класс на основе другого класса, применяется наследование. Выделим из синтаксиса описания класса часть, связанную с наследованием:

```
class <идентификатор> [<параметры типа>] [: <список наследования>]

<список наследования> :: {<тип класса> | <тип интерфейса>} [, ...]
```

Если список наследования не указан, то класс неявно наследуется от класса System.Object. В списке наследования может быть только один базовый класс, но неограниченное количество интерфейсов.

Базовый класс не должен быть классом System.Array, System.Delegate, System.MulticastDelegate, System.Enum или System.ValueType. Кроме того, объявление универсального класса (см. § 5.1) не может использовать System.Attribute в качестве базового класса.

4.6.1. Свойства наследования

Класс наследует члены своего прямого базового класса. Наследование означает, что класс неявно содержит все члены его прямого базового класса, исключая конструкторы экземпляров, деструкторы и статические конструкторы базового класса. Важные аспекты наследования:

• Наследование транзитивно. Если C произведено из B, а B произведено из A, то C унаследует как члены, объявленные в B, так и члены, объявленные в A;

• Производный класс расширяет свой прямой базовый класс. Производный класс может добавлять новые члены к своим потомкам, но он не может удалить определение унаследованного члена;

• Конструкторы экземпляров, деструкторы и статические конструкторы не наследуются, но все другие члены, независимо от их объявленной доступности, наследуются. Однако, в зависимости от их объявленной доступности, унаследованные члены могут быть недоступны в производном классе;

• Производный класс может скрывать унаследованные члены путем объявления новых членов с тем же именем или сигнатурой. Скрытие унаследованного члена не приводит к удалению данного члена, оно просто приводит к тому, что этот член становится недоступен непосредственно из произ-

водного класса;

• Экземпляр класса содержит неявное преобразование из типа производного класса в любой из его типов базового класса. Таким образом, ссылка на экземпляр некоторого производного класса может обрабатываться как ссылка на экземпляр какого-либо из его базовых классов;

• Класс может объявить абстрактные или виртуальные методы, свойства и индексаторы, а производный класс может переопределить реализацию данных членов функций. Это позволяет классам реализовывать полиморфное поведение.

В отличие от языка C++, множественное наследование в языке C# не поддерживается, а структуры вообще нельзя наследовать. Класс или структура может реализовывать несколько интерфейсов, но такой вид наследования существенно отличается от множественного наследования классов.

Пример:

```
class X { }
class Y { }
class Z1 : X, Y { } // Ошибка

interface IMy1 { }
interface IMy2 { }
class Z2 : X, IMy1, IMy2 { } // OK
```

Запрещено циклическое наследование. Так, если класс A наследуется от B, а класс B наследуется от C, то класс C не может наследоваться от A. Пример:

```
class A1: B1 { }
class B1: C1 { }
class C1: A1 { } // Ошибка

class A2 : B2.C2 { }
class B2 : A2 // Ошибка
{
      public class C2 { }
}

class A3
{
      class B3 : A3 { } // OK
}
```

 Пример: Samples\4.6\4_6_1_inherit.

4.6.2. Доступ к членам при наследовании

4.6.2.1. Модификаторы доступа

Производный класс имеет доступ к членам базового класса, которые:

• Объявлены с модификатором **public** – без ограничений;

• Объявлены с модификатором **protected** или **protected internal** – без ограничений;

• Объявлены с модификатором **internal** – если базовый класс описан в той же программе, что и производный.

К полям с модификатором **private** доступа нет, хотя они могут опосредованно вызываться при обращении к другим членам базового класса (методам, свойствам и т.д.).

Пример:

```
class MyBaseClass1
{
      public int A = 0;
      internal int B = 1;
      protected internal int C = 2;
      protected int D = 3;
      private int E = 4;
}

class MyDerClass1 : MyBaseClass1
{
      void F()
      {
            A = 5;
            B = 5;
            C = 5;
            D = 5;
            E = 5; // Ошибка
      }
}
```

4.6.2.2. Сокрытие членов

Если сигнатура члена производного класса, допускающего перегрузку, совпадает с сигнатурой члена в производном классе, компилятор выдает предупреждение о сокрытии наследуемого члена. Из этой ситуации есть два выхода:

1) Указать у данного члена в производном классе модификатор **new**. Тем самым мы покажем компилятору, что сокрытие совершено сознательно. Соответствующий член базового класса нам больше не нужен, вместо него

определяется новый член.

2) Если соответствующий член базового класса является виртуальным, указать модификатор **override**. Тогда новый член перегрузит член базового класса (см. § 4.7).

Если же сигнатуры членов не совпадают, то единственный способ избавиться от предупреждений компилятора – использовать модификатор **new**. Или дать члену в производном классе другое имя. Пример:

```
abstract class MyBaseClass2
{
      abstract public void A();
      virtual protected void B() { }
      virtual public void C() { Console.WriteLine("B.C"); }
      virtual public void D() { Console.WriteLine("B.D"); }
      virtual public void E() { Console.WriteLine("B.E"); }

      public void M1() { }
      private void M2() { }
      protected struct M3 { }
}

class MyDerClass2 : MyBaseClass2
{
      public override void A() { }
      // Предупреждение
      // void B() { }
      // Ошибка
      // public override void B() { }
      new void C() { Console.WriteLine("D.C"); }
      new public void D() { Console.WriteLine("D.D"); }
      override public void E() { Console.WriteLine("D.E"); }

      new public void M3() { }
      private void M2() { }
      new protected struct M1 { }
}

static int Main()
{
      MyDerClass2 der = new MyDerClass2();
      der.C(); // B.C
      ((MyBaseClass2)der).C(); // B.C
      der.D(); // D.D
      ((MyBaseClass2)der).D(); // B.D
      der.E(); // D.E
      ((MyBaseClass2)der).E(); // D.E
      return 0;
}
```

Предупреждение для «**void** B()» связано как раз с тем, что в базовом классе тоже есть член B, но ни модификатор **new**, ни модификатор **override** не указаны. Хотя в данном случае **override** указывать нельзя, т.к. сигнатуры членов не совпадают. Правила для перегрузки виртуальных методов очень

строгие – не должны различаться даже уровни доступа (в отличие от языка C++, например). С этим связана ошибка для «**public override void** B()» – в базовом классе указан другой уровень доступа **protected**. При выводе на консоль метод C() всегда вызывается для базового класса, потому что в производном классе он закрыт (**private**). Метод D() мы можем вызвать как для базового (используя операцию преобразования типа), так и для наследуемого класса. Метод E() виртуальный, поэтому всегда вызывается для класса-потомка.

Для членов M1 и M3 предупреждения не показываются, т.к. указан модификатор **new**. Иначе компилятор бы отобразил их, несмотря на то, что сигнатуры у них совершенно разные – один из членов является методом, а другой – структурой. Для M2 предупреждение не отображается, т.к. член является закрытым (**private**) в базовом классе. Следовательно, класс-потомок никак не может получить к нему доступ, и смысл перекрытия теряется.

4.6.2.3. Доступ к конструктору базового класса

В языке C++ доступ к конструктору базового класса осуществлялся следующим образом:

```
class MyClass1
{
public:
      MyClass1( /* список формальных параметров */ ) {}
};

class MyClass2 : public MyClass1
{
public:
      MyClass2( /* список формальных параметров */ ) :
            MyClass1( /* список фактических параметров */ ) {}
};
```

В языке C# вместо имени базового класса используется ключевое слово **base** (см. пункт 4.4.2). Как и в языке C++, конструктор по умолчанию (если он определен в базовом классе, или если в базовом классе не определен ни один конструктор) указывать не обязательно. Если в базовом классе есть конструкторы с параметрами, и нет конструктора по умолчанию, в производном классе обязательно должны быть конструкторы, явно обращающиеся к конструкторам базового класса.

Пример:

```
class MyBaseClass3
```

```
{
      public MyBaseClass3(int i) { }
}

class MyDerClass31 : MyBaseClass3 // Ошибка
{
}

class MyDerClass32 : MyBaseClass3 // OK
{
      protected MyDerClass32() : base(5) { } // OK
      protected MyDerClass32(string s) { } // Ошибка
}

class MyBaseClass4
{
      public MyBaseClass4() { }
      public MyBaseClass4(int i) { }
}

class MyDerClass41 : MyBaseClass4 // OK
{
}

class MyDerClass42 : MyBaseClass4 // OK
{
      public MyDerClass42() : base(5) { } // OK
      public MyDerClass42(string s) { } // OK
}
```

 Пример: Samples\4.6\4_6_2_access.

4.6.3. Абстрактные классы

Как видно из примера выше, класс, содержащий хотя бы один абстрактный член (объявленный в нем самом или базовом классе), также должен быть абстрактным. Хотя обратное утверждение неверно – в абстрактном классе может не быть абстрактных членов. Абстрактные члены не имеют тела и не могут быть закрытыми (**private**) – иначе как класс-потомок сможет их перегрузить?

Запрещено создавать экземпляры абстрактных классов. Абстрактный класс – это некоторый «набросок», довершать картину которого будут классы-потомки. Мы уже знакомы с абстрактными классами файловых потоков, но есть множество других.

Пример:

```
class C1
{
      abstract void F(); // Ошибка
      protected abstract void F() { } // Ошибка
}
```

```
class C2 // Ошибка
{
      protected abstract void F();
}

abstract class C3
{
      protected abstract void F(); // OK
}

class C4 : C3 // Ошибка
{
}

class C5 : C3 // OK
{
      protected override void F() { } // OK
}

abstract class C6 : C3 // OK
{
      abstract C6(); // Ошибка
      abstract ~C6(); // Ошибка
}

abstract class C7 { }
class C8 : C7 { }

static int Main()
{
      C7 x = new C7(); // Ошибка
      C8 x = new C8();

      return 0;
}
```

Класс C7 описан как абстрактный, хотя не содержит абстрактных членов.

Конструкторы и деструкторы, в отличие от других методов, не могут быть абстрактными.

 Пример: Samples\4.6\4_6_3_abstract.

4.6.4. Изолированные классы

Если мы хотим быть уверенными, что класс никогда не будет использован как базовый, необходимо объявить его как изолированный (запечатанный) класс. При этом применяется модификатор **sealed**. Единственное ограничение: абстрактный класс не может быть изолированным, так как в силу своей природы абстрактные классы предназначены для использования в качестве базовых классов. Хотя изолированные классы предназначены для

предотвращения непреднамеренного создания производных классов, определение класса как изолированного позволяет выполнить оптимизацию периода выполнения. В частности, поскольку компилятор гарантирует отсутствие у класса любых производных классов, есть возможность преобразования виртуальных вызовов для изолированного класса в невиртуальные вызовы.

Также изолированный класс не может объявлять новые виртуальные члены. Пример:

```
class C1
{
      public virtual void F() { }
}

sealed class C2 : C1  // OK
{
      public virtual void F2() { } // Ошибка
      public override void F() { } // OK
}

abstract class C3 // OK
{
      public abstract void F(); // OK
}

sealed class C4 : C3 // Ошибка
{
}

class C5 : C3 // OK
{
      public override void F() { } // OK
}
```

 Пример: Samples\4.6\4_6_4_sealed.

§ 4.7. Перегрузка и полиморфизм

Перегрузка методов позволяет многократно использовать одни и те же имена методов, меняя лишь передаваемые аргументы. А все члены класса, объявленные в одном из базовых классов как абстрактные или виртуальные, могут быть перегружены в классе-потомке. Кроме того, в любом классе или структуре можно переопределить некоторые стандартные операторы выражений, чтобы они научились работать с экземплярами данного класса или структуры.

4.7.1. Статический полиморфизм

Перегрузка полезна, по крайней мере, в трех сценариях. Первый: нам нужно иметь единое имя метода, поведение которого немного различается в зависимости от типа переданных аргументов. Второй сценарий, в котором выгодно применять перегрузку метода, – использование конструкторов, которые, в сущности, представляют собой методы, вызываемые при создании экземпляра объекта. Допустим, мы хотите создать класс, который может быть построен несколькими способами. Например, он использует описатель (**int**) или имя (**string**) файла, чтобы открыть его. Поскольку правила C# диктуют, что у конструктора класса должно быть такое же имя, как и у самого класса, мы не можете просто создать разные методы для переменных каждого типа. Вместо этого нужно использовать перегрузку конструктора. Третий сценарий – перегрузка индексатора. Т.к. он имеет фиксированное имя **this**, перегрузка является единственным вариантом для определения в классе нескольких индексаторов. Такой вариант перегрузки иногда называют статическим полиморфизмом.

О перегрузке метода важно помнить следующее: список аргументов каждого метода должен отличаться. При этом модификаторы **ref** и **out** не различаются. Если при перегрузке возникают конфликты с членами базового класса, как мы уже отмечали, необходимо использовать модификатор **new**.

Пример:

```
class MyClass
{
      public MyClass(int i) { } // OK
      public MyClass(string s) { } // OK
```

```
        void F(int x, ref double y) { } // OK
        int F(int x, ref double y) { return 0; } // Ошибка
        void F(out int x, out double y) { x = 1; y = 1; } // OK
        void F(int x, out double y) { } // Ошибка
        void F(int x, ref int y) { } // OK

        public int this[int i] { get { return 0; } } // OK
        public int this[string s] { get {return 0; } } // OK
        public int this[params int[] a] { get { return 0; } } // OK
}
```

 Пример: Samples\4.7\4_7_1_stat.

4.7.2. Виртуальный полиморфизм

В отличие от статического полиморфизма, виртуальный полиморфизм работает несколько по-другому. Перегрузка члена класса с помощью ключевого слова **new** возможна, если у нас есть ссылка на экземпляр производного класса. А что будет, если есть ссылка, приведенная к типу базового класса, но нужно, чтобы компилятор вызывал реализацию метода из производного класса? Это именно тот момент, когда в дело вступает виртуальный полиморфизм. Он позволяет переопределить часть функциональности базового класса в классах-потомках.

Виртуальный полиморфизм позволяет многократно определять члены в иерархии классов так, что в период выполнения в зависимости от того, какой именно объект используется, происходит обращение к соответствующей версии данного члена.

 Пример: Samples\4.7\4_7_2_virt.

4.7.2.1. Перегрузка методов

Для объявления виртуального метода в базовом классе следует использовать модификаторы **virtual** и **abstract**. В первом случае виртуальный метод должен содержать тело метода (если он не является внешним), во втором случае имеем абстрактный метод. Он не может иметь тела и не может являться внешним (**extern**). Допускается объявление абстрактных членов только в абстрактных классах. При перегрузке метода применяется модификатор **override**, причем сигнатуры перегружаемого и перегружающего методов должны полностью совпадать (тип возвращаемого значения, список формальных параметров, дополнительные модификаторы). Пример:

```
    abstract class AbsBase
```

```
{
	protected abstract int F1();
	public virtual int F2() { return 0; }
	private virtual int F3() { return 0; } // Ошибка
	public virtual extern void F4();
	public virtual int F5() { return 0; }
}

class Base : AbsBase
{
	public override int F1() { return 1; } // Ошибка
	protected override int F1() { return 1; }
	public override int F2() { return 1; }
}

class Child : Base
{
	protected override void F1() { return 2; } // Ошибка
	public override int F2() { return 2; }
	public new int F5() { return 2; }
}
```

По сравнению с языком C++, в синтаксис введены некоторые ограничения. Во-первых, виртуальные и абстрактные методы нельзя объявлять как закрытые (AbsBase.F3 в примере). В принципе, это не влияет на функциональность, поскольку такие методы по определению не будут видимы в производных классах. Во-вторых, для таких методов нельзя изменять модификатор доступа при перегрузке (Base.F1 в примере). В принципе, ужесточение уровня доступа к виртуальному методу в производном классе не имеет смысла, т.к. мы всегда можем преобразовать ссылку на производный класс к ссылке на базовый класс, в котором этот метод имеет более доступный модификатор. Пример для C++:

```
class Base
{
public:
	virtual void F(void) { return 0; }
};

class Child : public Base
{
protected:
	virtual void F(void) { return 1; }
};

int main(void)
{
	Child x;
	x.F(); // Ошибка
	((Base &)x).F(); // Но так ошибки нет!
	return 0;
}
```

Однако смягчение уровня доступа иногда было полезно. Тем не менее, разработчики языка C# отказались и от этого.

В этом примере при наследовании был использован модификатор **public**. Мы помним, что в языке C++ при наследовании можно было использовать и другие модификаторы доступа, а по умолчанию использовался **private**. В языке C# от этого отказались, модификатор не указывается, а подразумевается **public**. Это логично, другие, более защищенные, модификаторы доступа не имеют особого смысла. По показанной выше схеме всегда можно преобразовать ссылку к типу базового класса, где уровень защиты ниже.

Эффект от перегрузки заключается в том, что при вызове, например, метода F2 для экземпляра класса Base всегда будет вызываться метод Base.F2, а для класса Child – Child.F2, независимо от преобразований типов. Метод F5 в классе Base не переопределен, поэтому для экземпляра этого класса будет вызываться AbsBase.F5. А в классе Child определен новый метод F5, не имеющий отношения к виртуальному методу AbsBase.F5. Поэтому здесь то, какой метод будет вызван, уже зависит от типа ссылки. Сравните:

```
Base b = new Base();
Child c = new Child();

Console.WriteLine(b.F2()); // 1
Console.WriteLine(b.F5()); // 0
Console.WriteLine(((AbsBase)b).F2()); // 1
Console.WriteLine(((AbsBase)b).F5()); // 0
Console.WriteLine(c.F2()); // 2
Console.WriteLine(c.F5()); // 2
Console.WriteLine(((Base)c).F2()); // 2
Console.WriteLine(((Base)c).F5()); // 0
Console.WriteLine(((AbsBase)c).F2()); // 2
Console.WriteLine(((AbsBase)c).F5()); // 0
```

Другой пример:

```
static void F(object obj)
{
      Console.WriteLine(obj.GetType());
}

static int Main()
{
      Child c = new Child();

      F(c); // VirtualPolymorphismSample.Program+Child
      F(1); // System.Int32
      F("abc"); // System.String
      return 0;
}
```

Если бы метод GetType() не был виртуальным, на консоли всегда бы

отображался текст «System.Object».

4.7.2.2. Доступ к виртуальному методу базового класса

В языке C++ мы могли получить доступ ко всей цепочке виртуальных методов, причем как из класса-потомка, так и из внешнего класса или процедуры (если это позволял модификатор доступа). Делалось это при помощи преобразования указателя или ссылки на экземпляр класса (как в примере выше), или при помощи квалификатора области действия

```
<имя класса>::<член>
```

Так, вместо записи «`((Base &)x).F()`» можно использовать запись «`x.Base::F()`». При этом можно указывать не только непосредственные базовые классы, но и любые другие базовые классы вверх по иерархии. В языке C# такой возможности нет. Извне класса мы вообще никак не можем повлиять на цепочку вызовов виртуальных методов, а в самом классе можем вызывать виртуальные методы лишь непосредственных предков класса, используя уже знакомое ключевое слово **base**:

```
base.<член>
```

Таким же образом можно получить доступ к другим, не виртуальным членам базового класса. Пример:

```
class Child : Base
{
      public override int F2() { return 2; }
      public new int F5() { return 2; }
      public int F6() { return F2() + base.F2(); }
}

static int Main()
{
      Child c = new Child();

      Console.WriteLine(c.F6()); // 3
      return 0;
}
```

4.7.2.3. Перегрузка свойств и индексаторов

У свойств и индексаторов, как и методов, могут быть указаны модификаторы **virtual**, **override** или **abstract**. Это позволяет производным классам наследовать и перегружать свойства подобно любому другому члену, унаследованному от базового класса. При этом, если в базовом классе есть оба

метода – получатель и установщик – при перегрузке одного из них нужно также перегружать и второй.

Данные модификаторы указываются для всего свойства или индексатора, а не для его методов доступа. У абстрактного свойства или индексатора, в отличие от абстрактных методов, должно быть тело – в нем нужно определить, какие предусмотрены методы доступа. Методы доступа в абстрактном свойстве или индексаторе тел не имеют. При доступе к индексатору базового класса используется синтаксис

```
base"["<список фактических аргументов>"]"
```

В остальном действуют те же правила, что и при перегрузке методов. Пример:

```
abstract class AbsBase
{
      public virtual int Prop { get { return 0; } }
      public abstract string this[int i]
      {
            get;
      }
}

class Base : AbsBase
{
      public override int Prop { get { return base.Prop + 1; } }
      public override string this[int i]
      {
            get { return "Base." + i; }
      }
}

class Child : Base
{
      public override int Prop { get { return base.Prop + 1; } }
      public override string this[int i]
      {
            get { return "Child." + base[i]; }
      }
}

static int Main()
{
      Base b = new Base();
      Child c = new Child();

      Console.WriteLine(b.Prop); // 1
      Console.WriteLine(b[5]); // Base.5
      Console.WriteLine(c.Prop); // 2
      Console.WriteLine(c[7]); // Child.Base.7
      return 0;
}
```

4.7.2.4. Изолированные члены класса

Изолированный (или запечатанный член класса) должен быть объявлен с модификаторами **override sealed** (т.к. если член не перегружается, то нет смысла в его изоляции). После этого он перестает быть виртуальным в классах-потомках.

Пример:

```
class Child : Base
{
      public override sealed int F2() { return 2; }
}

class Child2
{
      public override int F2() { return 3; } // Ошибка
}
```

Модификатор **sealed** следует применять в тех случаях, когда мы по каким-либо причинам хотим запретить дальнейшую перегрузку члена.

4.7.3. Перегрузка операторов

Перегрузка операторов позволяет переопределить стандартные операторы выражений языка C# для применения их к типам, определенным пользователем.

Когда мы изучали операторы выражений, то видели, что все они применимы лишь для встроенных типов данных по значению. Для других типов данных они работать не будут. Например:

```
int[] a = { 1, 2, 3 };
int[] b = { 4, 5, 6 };
int[] c = a + b; // Ошибка
```

Однако, некоторые структуры (DateTime) и типы по ссылке (String) переопределяли ряд операторов таким образом, что их можно было использовать для аргументов данных типов. Мы можем поступить так же.

 Пример: Samples\4.7\4_7_3_oper.

4.7.3.1. Ограничения синтаксиса

С точки зрения класса, описание перегруженного оператора выглядит похоже на описание метода:

```
<перегрузка оператора> :: [<атрибуты>] public static [extern]
```

```
            <описание оператора> <тело оператора>

      <описание оператора> :: <унарный оператор>
      <описание оператора> :: <бинарный оператор>
      <описание оператора> :: <оператор преобразования типа>

      <унарный оператор> :: <тип> operator <оператор> (<тип1>
<идентификатор1>)
      <бинарный оператор> :: <тип> operator <оператор> (<тип1>
<идентификатор1>, <тип2> <идентификатор2>)
      <оператор преобразования типа> :: {implicit | explicit} operator <тип>
(<тип1> <идентификатор1>)

      <тело оператора> :: <блок>
      <тело оператора> :: ;
```

Все параметры операторов должны быть параметрами-значениями. Любые модификаторы параметров недопустимы.

При этом язык C# относится к перегрузке операций более строго, чем язык C++. Не останавливаясь на подробном сравнении подходов к перегрузке операторов в этих языках, рассмотрим ограничения, налагаемые на перегрузку операторов компилятором языка C#:

1) Из унарных можно перегрузить только операторы «+», «-», «!», «~», «++», «--», «true» и «false»;

2) Из бинарных можно перегрузить арифметические операторы «+», «-», «*», «/», «%», побитовые операторы «&», «|», «^», «<<», «>>» и операторы сравнения «==», «!=», «>», «<», «>=», «<=»;

3) Все методы, представляющие перегружаемые операторы, должны иметь модификаторы **public** и **static**;

4) Нельзя перегрузить оператор индексации «[]», но вместо этого в классе можно определить индексатор, обладающий даже более широкими возможностями;

5) Нельзя перегрузить оператор присваивания «=». Он реализован на системном уровне, и обеспечивает побитовое копирование для типов по значению и копирование ссылок для ссылочных типов. Также запрещена перегрузка составных операторов присваивания. Однако, если в классе определен разрешенный для перегрузки арифметический или побитовый оператор «op», то составной оператор «op=» определяется автоматически;

6) Операторы, которые в настоящее время в языке C# не определены, также не перегружаются. Например, мы не можем определить оператор «**» как разновидность возведения в степень, поскольку в языке C# не определен оператор «**»;

7) Нельзя изменить синтаксис операторов. Мы не можем изменить бинарный оператор «*» так, чтобы он принимал три аргумента, поскольку его синтаксис по определению подразумевает два аргумента;

8) Приоритет операторов при перегрузке не изменяется.

Дополнительные ограничения приведены в дальнейшем материале.

4.7.3.2. Унарные операторы

К переопределенным в классе T унарным операторам применяются следующие дополнительные ограничения:

• унарные операторы «+», «-», «!» или «~» должны принимать единственный параметр типа T или T? и могут возвращать любой тип;

• унарные операторы «++» или «--» должны принимать единственный параметр типа T или T? и должны возвращать тип T или тип, производный от него;

• унарные операторы «true» или «false» должны принимать единственный параметр типа T или T? и должны возвращать тип **bool**. При этом эти операторы перегружаются только попарно.

Например, определим класс, содержащий вектор с целочисленными координатами. Операторы инкремента и декремента будут увеличивать или уменьшать количество компонентов в векторе. Также определим ряд конструкторов, метод ToString(), свойство, возвращающее размер вектора и индексатор для доступа к компонентам вектора:

```
class Vector
{
      private int[] V;

      public int Length
      {
            get { return V == null ? 0 : V.Length; }
      }

      public int this[int i]
      {
            get { return V[i]; }
            set { V[i] = value; }
      }

      public Vector()
      {
            V = null;
      }

      public Vector(int size)
```

```
        {
            V = new int[size];
        }

        public Vector(int[] arr)
        {
            V = (int[])arr.Clone();
        }

        public override string ToString()
        {
            string rez = "[ ";
            for (int i = 0; i < Length; i++) rez += V[i] + " ";
            return rez + "]";
        }

        void Copy(Vector v, int pos, int len)
        {
            if (v.Length > 0) Array.Copy(v.V, 0, V, pos, len);
        }

        public static Vector operator ++(Vector v)
        {
            Vector rez = new Vector(v.Length + 1);
            rez.Copy(v, 0, v.Length);
            return rez;
        }

        public static Vector operator --(Vector v)
        {
            Vector rez = new Vector(v.Length - 1);
            rez.Copy(v, 0, rez.Length);
            return rez;
        }
    }

    static int Main()
    {
        int[] a = { 1, 2, 3 };
        int[] b = { 4, 5, 6 };

        Vector v1 = new Vector(a);
        Vector v2 = new Vector(b);
        v1--;
        v2++;
        Console.WriteLine("v1 = " + v1); // v1 = [ 1 2 ]
        Console.WriteLine("v2 = " + v2); // v2 = [ 4 5 6 0 ]
        return 0;
    }
```

Обратите внимание на следующие аспекты:

1) Мы не указываем, какую именно форму инкремента и декремента перегружаем (префиксную или постфиксную). Это определяется из контекста применения оператора. При этом операторы ведут себя естественно:

```
    Vector vtest1 = v1++;
    Vector vtest2 = --v2;
    Console.WriteLine("vtest1 = " + vtest1); // vtest1 = [ 1 2 ]
```

```
      Console.WriteLine("v1 = " + v1); // v1 = [ 1 2 0 ]
      Console.WriteLine("vtest2 = " + vtest2); // vtest2 = [ 4 5 6 ]
      Console.WriteLine("v2 = " + v2); // v2 = [ 4 5 6 ]
      Console.WriteLine();
```

Т.е. результатом выражения «v1++» будет старое значение вектора v1, а результатом «--v2» – новое значение v2.

2) Не следует в данных операторах модифицировать экземпляр объекта, передаваемого в качестве параметра. Фактически модификация значения операнда нарушала бы их стандартную семантику.

4.7.3.3. Бинарные операторы

К переопределенным в классе T бинарным операторам применяются следующие дополнительные ограничения:

• бинарные операторы побитового сдвига «<<» или «>>» должны принимать два параметра, первый из которых должен иметь тип T или T?, а второй – **int** или **int**?, и может возвращать любой тип;

• остальные бинарные операторы должны принимать два параметра, по крайней мере один из которых должен иметь тип T или T?, и может возвращать любой тип;

• следующие операторы должны быть объявлены попарно: «==» и «!=», «>» и «<», «>=» и «<=». Для каждого объявления одного оператора из пары должно быть соответствующее объявление другого оператора из пары. Два объявления операторов соответствуют, если у них одинаковый тип возвращаемого значения и одинаковый тип каждого параметра.

Например, перегрузим в классе Vector операторы сложения и вычитания векторов:

```
      void CheckSize(Vector v)
      {
            if (Length != v.Length)
            {
                  throw new ArgumentException("Размеры векторов не
совпадают!");
            }
      }

      public static Vector operator +(Vector v1, Vector v2)
      {
            v1.CheckSize(v2);
            Vector rez = new Vector(v1.V);
            for (int i = 0; i < rez.Length; i++) rez[i] += v2[i];
            return rez;
      }
```

```
      public static Vector operator -(Vector v1, Vector v2)
      {
            v1.CheckSize(v2);
            Vector rez = new Vector(v1.V);
            for (int i = 0; i < rez.Length; i++) rez[i] -= v2[i];
            return rez;
      }
```

Если размеры векторов не совпадают, генерируем исключительную ситуацию. Как уже отмечалось, теперь для векторов можно также применять операции «+=» и «–=»:

```
      Console.WriteLine("v1 + v2 = {0}", v1 + v2); // v1 + v2 = [ 5 6 7 ]
      v1 -= v2;
      Console.WriteLine("v1 = {0}", v1); // v1 = [ -3 -3 -6 ]
```

4.7.3.4. Операторы преобразования типов

Ранее упоминалось, что оператор «`()`», применяемый при приведении типов, не может быть перегружен, и вместо этого используется нестандартное преобразование.

Объявление оператора преобразования, включающее зарезервированное слово **implicit**, вводит неявное, а **explicit** – явное преобразование, определенное пользователем. Оператор преобразования преобразует от исходного типа, указанного типом параметра оператора преобразования, к конечному типу, указанному типом возврата оператора преобразования.

Для данного исходного типа S и конечного типа T, если S или T являются типами, допускающими присваивание **null**, разрешено объявлять преобразование типа, если:

• S и T являются разными типами;

• либо S, либо T является типом структуры или класса, где имеет место объявление этого оператора;

• ни S, ни T не является типом интерфейса;

• без преобразований, определенных пользователем, не существует преобразование от S к T или от T к S.

Примеры неправильных описаний операторов:

```
      class TypeCast
      {
            public static implicit operator TypeCast(TypeCast c) { return
null; }
            public static implicit operator Vector(int i) { return null; }
            public static implicit operator TypeCast(ICloneable c) { return
null; }
```

```
        public static implicit operator Object(TypeCast c) { return null;
}
    }
```

Например, пусть вектор при явном преобразовании к числу с плавающей точкой возвращает свою длину, а при неявном преобразовании целого числа к вектору получим вектор из одного компонента:

```
    public static explicit operator double(Vector v)
    {
        double len = 0;
        for (int i = 0; i < v.Length; i++) len += v[i] * v[i];
        return Math.Sqrt(len);
    }

    public static implicit operator Vector(int i)
    {
        Vector rez = new Vector(1);
        rez[0] = i;
        return rez;
    }
```

Пример использования:

```
    Console.WriteLine("|v1| = {0}", (double)v1); // |v1| = 7,348...
    v2 = 5;
    Console.WriteLine("v2 = {0}", v2); // v2 = [ 5 ]
```

4.7.3.5. Дополнительные возможности

Перегрузка логических операторов

Как уже было отмечено, напрямую перегрузить условные логические операторы «`&&`» и «`||`» нельзя. Однако, компилятор может вычислить выражения «`x && y`» и «`x || y`» для операндов пользовательского типа T, если:

• В классе T определены операторы «`&`» и «`|`», причем для двух операндов типа T и с результатом типа T;

• Класс T должен содержать объявления операторов «`true`» и «`false`».

Если эти условия выполняются, вычисление условных логических операций происходит следующим образом:

```
    x && y = T.false(x) ? x : T.&(x, y)
    x || y = T.true(x) ? x : T.|(x, y)
```

В обеих этих операциях выражение «x» вычисляется только один раз, а выражение «y» либо не вычисляется, либо также вычисляется один раз.

Например, добавим в класс вектора оператор «`&`», соединяющий два вектора в один, а также операторы «`true`» и «`false`». Будем считать вектор

истинным, если он не пуст:

```
	public static Vector operator &(Vector v1, Vector v2)
	{
		Vector rez = new Vector(v1.Length + v2.Length);
		rez.Copy(v1, 0, v1.Length);
		rez.Copy(v2, v1.Length, v2.Length);
		return rez;
	}

	public static bool operator true(Vector v)
	{
		return v.Length != 0;
	}

	public static bool operator false(Vector v)
	{
		return v.Length == 0;
	}
```

Теперь к вектору можно применять оператор «&» для конкатенации и «&&», который вернет x, если вектор x пустой, и конкатенацию векторов x и y в противном случае:

```
	Vector v3 = v2 & v1;
	Vector v4 = new Vector();

	Console.WriteLine("v2 & v1 = {0}", v3); // v2 & v1 = [ 5 -3 -3 -6 ]
	Console.WriteLine("v1 && v4 = {0}", v1 && v4); // v1 && v4 = [ -3 -3 -6
]
	Console.WriteLine("v4 && v1 = {0}", v4 && v1); // v4 && v1 = [ ]
```

В принципе, обычно операторы «&&» и «||» применяются для составления условий (в тернарном или условном операторе, а также в операторах цикла). Результатом этих операций для векторов является вектор. Но в классе вектора определены операторы «true» и «false», а это означает, что экземпляры векторов можно использовать в условиях:

```
	bool f = true;

	if (v1 && v2) { } // OK
	if (v1 && v2 && f) { } // Ошибка
```

Однако, во втором условии есть ошибка, т.к. оператор «Vector && bool» не определен. Чтобы работала и такая запись, необходимо предусмотреть оператор неявного преобразования вектора к логическому типу или наоборот – логического типа к вектору. Остановимся на первом варианте:

```
	public static implicit operator bool(Vector v)
	{
		return v.Length != 0;
	}
```

После этого компиляция выполняется успешно.

Перегрузка операторов отношения

При перегрузке операторов «==» и «!=» имеет смысл также перегрузка метода Object.Equals(). В этом случае все методы, использующие вызов Equals, будут сравнивать не ссылки на класс (или не побитовое сравнение структур), а выполнять сравнение по определенному алгоритму.

Среда .NET требует, чтобы при перегрузке метода Object.Equals() обязательно перегружался метод Object.GetHashCode(). Данный метод играет роль хэш-функции для типа данных. Метод GetHashCode можно использовать в алгоритмах хэширования и таких структурах данных, как хэш-таблицы. В принципе, реализация метода GetHashCode по умолчанию не гарантирует уникальность возвращаемых для объекта значений. Более того, в платформе .NET Framework возвращаемые этим методом значения одинаковы в разных версиях .NET Framework. Следовательно, реализацию такого метода по умолчанию не следует использовать для хэширования в качестве уникального идентификатора объекта. А если задача хэширования перед нами не стоит, то в принципе не важно, какое значение возвращает данный метод. Можно, например, просто вернуть значение хэш-кода родительского класса.

Опишем эти методы и операторы в классе вектора:

```
public static bool operator ==(Vector v1, Vector v2)
{
      if (v1.Length != v2.Length) return false;
      for (int i = 0; i < v1.Length; i++)
      {
            if (v1[i] != v2[i]) return false;
      }
      return true;
}

public static bool operator !=(Vector v1, Vector v2)
{
      return !(v1 == v2);
}

public override bool Equals(object obj)
{
      return obj is Vector ? (Vector)obj == this : false;
}

public override int GetHashCode()
{
      return base.GetHashCode();
```

```
    }
```

Пример использования:

```
    Vector v5 = v2 & v1;

    Console.WriteLine(v3 == v5); // True
    Console.WriteLine(v3.Equals(v5)); // True
```

При перегрузке операторов «>», «<»,«>=», «<=» имеет смысл добавить в список наследования класса интерфейс IComparable (см. п. 4.9.4.1).

§ 4.8. Делегаты и события

Одно из полезных нововведений в языке C# – делегаты (delegates). Их назначение, по сути, совпадает с указателями на функции в языке C++. Однако, для того, чтобы хранить указатель на нестатический метод класса, одного указателя мало. Нужна пара указателей

```
{<адрес экземпляра класса>, <адрес метода>}
```

Один из них указывает на экземпляр класса, а второй – уже на сам метод. В момент вызова метода первый указатель должен инициализировать указатель **this**, после чего вызываемый метод может обрабатывать соответствующий экземпляр класса.

4.8.1. Предыстория вопроса

4.8.1.1. Сообщения в ООП

Острая необходимость в механизме указателей на методы класса возникла при появлении библиотек классов для Windows. Операционные системы DOS и Windows имеют диаметрально противоположный подход к организации общения программ с устройствами ввода-вывода. Рассмотрим пример – ожидание нажатия пользователем клавиши (при помощи функции kbhit() библиотеки консольного ввода-вывода языка C). Программа, написанная для ОС MS DOS, при этом входит в бесконечный цикл (рис. 4.7). Пока она работает, то постоянно потребляет аппаратные ресурсы, отнимая их у других программ, в принципе, ничего не делая в это время. Получается, что прикладная программа является центральным объектом, т.к. это она вызывает функции ОС, а не наоборот.

Программа, написанная для ОС MS Windows, во время ожидания команд от ОС ничего не делает. При этом не занимает ресурсы, требуемые для других программ и служб ОС, работающих в фоновом режиме. При поступлении команды от ОС, например, от устройства ввода, об этом оповещаются все заинтересованные программы (рис. 4.8).

Оповещение происходит посредством *механизма сообщений*. Сообщения рассылаются не всем программам. Например, сообщения о нажатии клавиш – только активным программам (имеющим *фокус ввода*) или программам, установившим перехват данных сообщений (hooks).

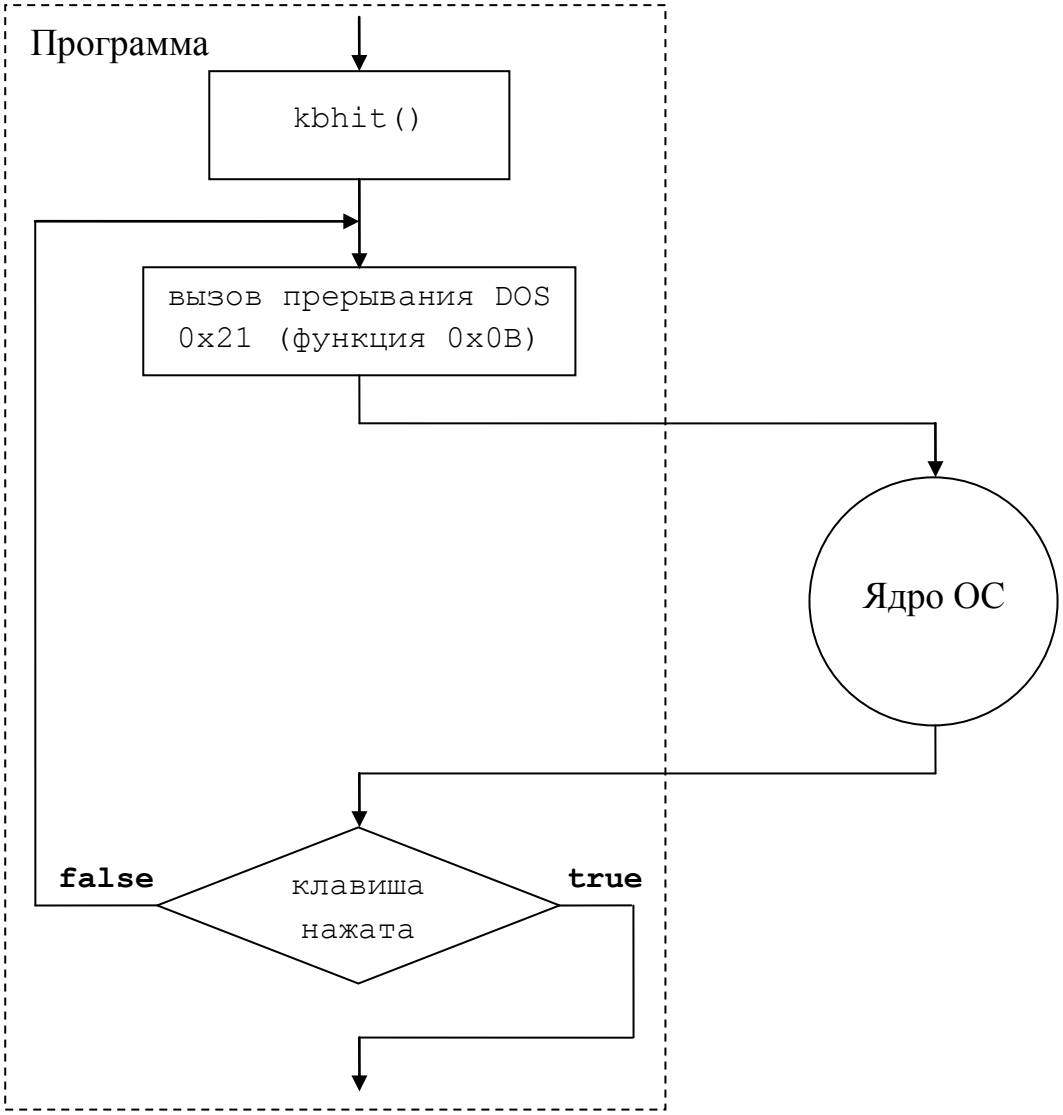


Рис. 4.7 – Схема работы программы для ОС MS DOS

В свою очередь, программа состоит из множества компонентов, и определенные сообщения нужны только некоторым компонентам. Поэтому любая программа Windows содержит *диспетчер сообщений*, отсылающий приходящие сообщения заинтересованным компонентам.

Учитывая сложность программного интерфейса (Application Programming Interface, API) Windows, разработчики сред программирования для этой ОС создали собственные объектно-ориентированные библиотеки, предоставляющие пользователям более простой доступ к возможностям Windows. Это библиотеки ATL/WTL/MFC (Microsoft), OWL/VCL (Borland) и т.д. непосредственно для Windows, и кросс-платформенные библиотеки LCL (Lazarus), CLX (Borland), Qt (Qt Software, Nokia) и т.д.

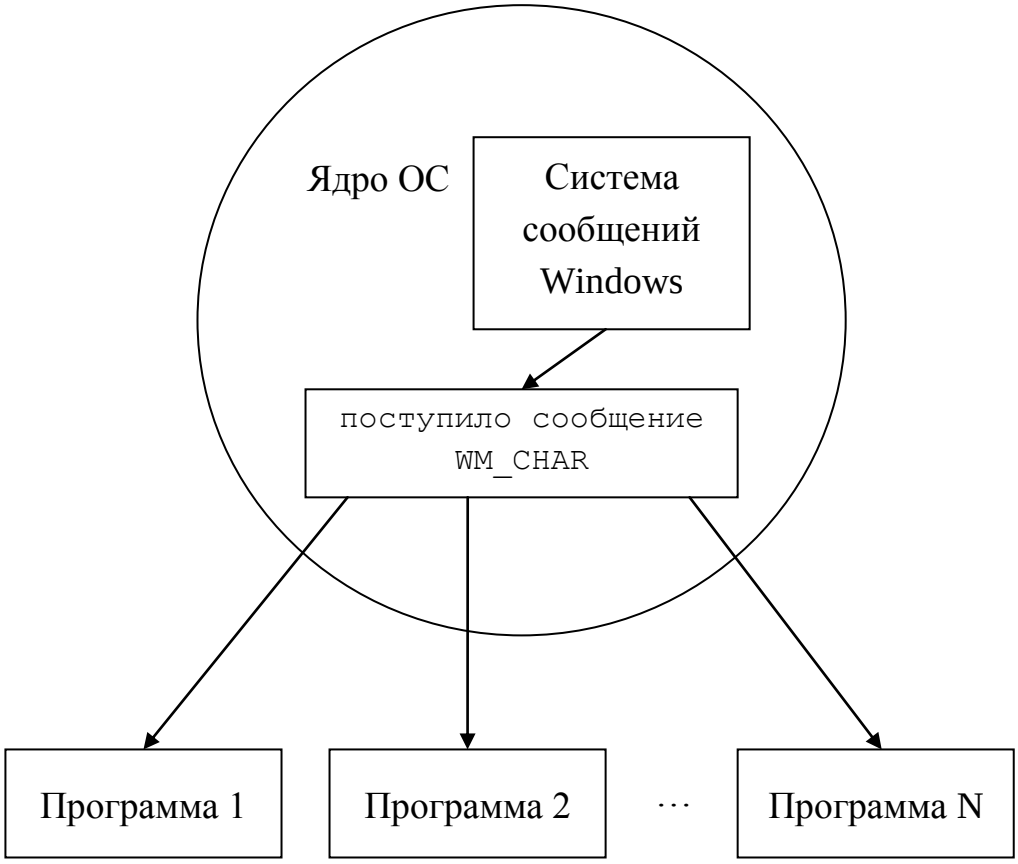


Рис. 4.8 – Схема работы программы для ОС MS Windows

Таким образом, отсылка сообщения компоненту, представленному экземпляром какого-либо класса, заключается в вызове метода этого класса. Как указать диспетчеру, какой метод и какого класса вызывать?

4.8.1.2. Указатели на методы класса

В языке C++ приходилось использовать различные ухищрения, чтобы реализовать универсальные указатели на методы (т.е. не зависящие от иерархии классов). Дело в том, что стандартным способом можно было объявить лишь указатель на метод фиксированного класса:

```
#include <stdio.h>

class MyClass1
{
public:
      void F(char *s) { printf("%s\n", s); }
};

class MyClass2
{
public:
      void F1(char *s)  { printf("%s::%s\n", s, s); }
```

```
        void F2(char *s)  { printf("%s\n", s); }
};

typedef void (MyClass1::*Method)(char *);

void main(void)
{
        MyClass1 cls1;
        MyClass2 cls2;
        Method m1, m2;
        m1 = MyClass1::F;
        m2 = MyClass2::F2; // Ошибка
        (cls1.*m1)("Привет!"); // Привет!
        (cls2.*m2)("Привет!"); // Ошибка
}
```

Да, в этом случае m1 и m2 – это пары указателей, но один из них хранит адрес таблицы виртуальных функций (VFT), а не экземпляра класса. Поэтому, во-первых, в момент вызова экземпляр класса нужно указывать явно. А во-вторых, указатель работает только с одним типом класса.

Конечно, чтобы не зависеть от типа класса, можно было объявить шаблон метода (класс или структуру, перегружающую оператор вызова «()» и хранящую адрес экземпляра):

```
template <class T> struct Method
{
        typedef void (T::*MethodPtr)(char *);
        T *object;
        MethodPtr method;
        Method(T *obj, MethodPtr met)
        {
                object = obj;
                method = met;
        }
        void operator () (char *param)
        {
                (object->*method)(param);
        }
};

void main(void)
{
        MyClass1 cls1;
        MyClass2 cls2;
        Method<MyClass1> m1(&cls1, MyClass1::F);
        Method<MyClass2> m2(&cls2, MyClass2::F2);
        m1("Привет!"); // Привет!
        m2("Привет!"); // Привет!
}
```

В этом случае все работает, но описанная структура заточена только для методов с определенной сигнатурой – тип возвращаемого параметра **void**, один формальный параметр типа **char** *. Для других сигнатур придется

писать новые структуры. А их может быть неограниченно много.

Классы библиотеки MFC решали эту задачу созданием карты сообщений. Однако, использование карт сообщений было неудобным. Они не являлись элементом языка, а представляли собой просто набор макросов, редактировать который вручную настоятельно не рекомендовалось. Таким образом, ни о какой прозрачности и корректности типов говорить не приходилось.

Как это было уже не раз, более удобное решение предложили разработчики Borland C++ Builder. Они расширили язык C++ новым ключевым словом **__closure**, позволяющим объявлять указатели на методы любых классов:

```
typedef void (__closure * Method)(char *);

void main(void)
{
      Method m1, m2;
      m1 = cls1.F;
      m2 = cls2.F2;
      m1("Привет!"); // Привет!
      m2("Привет!"); // Привет!
}
```

Разработчики .NET Framework, в свою очередь, оценили этот подход и реализовали в языке C# похожее решение – делегаты. Однако, учитывая, что C# чисто объектный язык, то и делегаты являются классами. Они гарантируют, что делегат указывает на допустимый объект, а это в свою очередь означает получение всех достоинств указателей на методы без связанных с этим опасностей, таких как применение недопустимых адресов или разрушение памяти других объектов.

В C# две основных области применения делегатов: методы обратного вызова и обработчики событий.

4.8.2. Методы обратного вызова

Методы обратного вызова (callback methods) повсеместно используются в Windows для передачи указателя на функцию в другую функцию, чтобы последняя могла вызвать первую (через переданный ей указатель). Любой программист Win32 API наверняка не раз пользовался ими. Так, функция EnumWindows перечисляет все окна верхнего уровня на экране и для каждого окна вызывает переданную ей по указателю функцию.

Синтаксис объявления делегата:

```
      <объявление делегата> :: [<атрибуты>] [<модификаторы>] delegate <тип
возвращаемого значения> <идентификатор> [<параметры типа>] ([<список
формальных параметров>]) [<ограничения параметров типа>];
```

1) Атрибуты рассмотрены в § 5.4;

2) Допустимы модификаторы **new**, **public**, **protected**, **internal** и **private**. По смыслу они аналогичны таковым для классов;

3) Мы объявляем новый тип данных (названием которого является указанный идентификатор) – указатель на метод класса, возвращающий значение указанного типа и имеющий указанный список формальных параметров;

4) Параметры типа и ограничения параметров типа будут рассмотрены в § 5.1.

При использовании типа делегата в члене класса уровень доступа члена должен быть не менее строгим, чем уровень доступа делегата.

Пример:

```
      public delegate void D1 (int i);
      protected delegate void D2(int i);

      public D1 d1; // OK
      public D2 d2; // Ошибка
      private void M(D2 d) { } // OK
```

 Пример: Samples\4.8\4_8_2_delegate.

4.8.2.1. Создание экземпляра и вызов делегата

Экземпляр делегата может ссылаться на:

• статический метод;

• экземпляр класса (который не может иметь значение **null**) и метод экземпляра;

• другой делегат.

Синтаксис:

```
      <создание делегата> :: <тип делегата> <идентификатор> = new <тип
делегата> (<выражение создания делегата>);
      <создание делегата> :: <тип делегата> <идентификатор> = <выражение
создания делегата>;

      <выражение создания делегата> :: [<класс>.]<статический метод>
      <выражение создания делегата> :: [<экземпляр класса>.]<метод экземпляра>
      <выражение создания делегата> :: <экземпляр делегата>
```

Класс или экземпляр класса не обязательно указывать, если делегат со-

здается в том же классе, где объявлен метод. При этом метод или экземпляр делегата M должны быть совместимы с типом создаваемого делегата D, а это означает, что:

• D и M имеют одинаковое число параметров, и каждый параметр в D имеет такие же модификаторы **ref** или **out**, как и соответствующий параметр в M;

• для каждого параметра значения (без модификатора **ref** или **out**) существует неявное преобразование ссылки из типа параметра в D в соответствующий тип параметра в M;

• для каждого параметра **ref** или **out** тип параметра в D такой же, как тип параметра в M;

• существует неявное преобразование ссылки из типа возвращаемого значения M в тип возвращаемого значения D.

Пример:

```
      public delegate object CloneDelegate(string i);
      public static string CloneStr1(string str) { return (string)str.Clone();
}
      public static string CloneStr2(object obj) { return (string)(obj as
string).Clone(); }
      public static object CloneStr3(string str) { return str.Clone(); }
      public static object CloneStr4(object obj) { return (obj as
string).Clone(); }

      public delegate int IntDelegate(ref int i, int j);
      public static int Sum1(int i, int j) { return i + j; }
      public static int Sum2(int i, ref int j) { return j += i; }
      public static int Sum3(ref int i, int j) { return i += j; }

      static int Main()
      {
            CloneDelegate cd1 = new CloneDelegate(CloneStr1); // OK
            CloneDelegate cd2 = new CloneStr2; // OK
            CloneDelegate cd3 = new CloneDelegate(CloneStr3); // OK
            CloneDelegate cd4 = new CloneStr4; // OK

            IntDelegate id1 = new IntDelegate(Sum1); // Ошибка
            IntDelegate id2 = new IntDelegate(Sum2); // Ошибка
            IntDelegate id3 = new IntDelegate(Sum3); // OK

            return 0;
      }
```

Вызывается делегат как обычный метод с совместимой сигнатурой. Попытка вызова экземпляра делегата, имеющего значение **null**, приводит к исключению типа System.NullReferenceException. Пример:

```
      class MyClass
      {
```

```
        public delegate int Operation (int x);
        public int Sqr(int x) { return x * x; }
        public static int Cube(int x) { return x * x * x; }
    }

    static int Main()
    {
        MyClass cls = new MyClass();
        MyClass.Operation op1 = new MyClass.Operation(cls.Sqr);
        MyClass.Operation op2 = MyClass.Cube;
        MyClass.Operation op3 = null;

        Console.WriteLine(op1(5)); // 25
        Console.WriteLine(op2(5)); // 125
        Console.WriteLine(op3(5)); // NullReferenceException
        return 0;
    }
```

Ну а теперь приведем пример метода обратного вызова:

```
    public delegate void EnumDel(int idx, ref string str);
    static void OutStrings(string[] lines, EnumDel ed)
    {
        for (int i = 0; i < lines.Length; i++)
        {
            if (ed != null) ed(i, ref lines[i]);
            Console.WriteLine(lines[i]);
        }
    }

    static int Main()
    {
        EnumDel myenum = delegate(int i, ref string s)
        {
            s = String.Format("{0:0#} {1}", i, s);
        };
        OutStrings(new string[] { "строка 1", "строка 2", "строка 3" },
myenum);
        return 0;
    }
```

Метод OutStrings выводит список строк на консоль, но перед этим вызывает метод обратного вызова, представленный делегатом. В данном случае метод обратного вызова используется для нумерации строк. Делегат представлен анонимной функцией, о них говорится ниже.

Вывод на консоль:

```
    00 строка 1
    01 строка 2
    02 строка 3
```

4.8.2.2. Операции с делегатами

Как уже упоминалось, к делегатам применимы операции «==», «!=», «=», «+», «+=», «–» и «–=». Первые две осуществляют сравнение делегатов,

третья – присваивание. Остальные операции позволяют управлять *списком вызова делегата*.

Общий предок для всех делегатов – класс System.Delegate. Однако создаваемые нами делегаты имеют базовый класс System.MulticastDelegate, являющийся потомком System.Delegate. Поэтому они обладают всеми свойствами обычных делегатов, и дополнительно могут создавать списки вызова. При вызове делегата вызываются все методы, чьи адреса есть в списке. Получить этот список можно при помощи вызова метода

```
Delegate[] MulticastDelegate.GetInvocationList();
```

Операторы сложения объединяют списки вызовов двух делегатов, операторы вычитания убирают из первого списка вызова элементы, имеющиеся во втором списке. Это соответствует операциям объединения и разности множеств.

Также от класса Delegate класс MulticastDelegate наследует некоторые полезные свойства:

```
MethodInfo Method;
object Target;
```

Первый из них содержит ссылку на метод, второй – на экземпляр класса, представленный делегатом.

Пример:

```
public delegate void Operation(int x);
static void Sqr(int x) { Console.WriteLine(x * x); }
static void Cube(int x) { Console.WriteLine(x * x * x); }

static int Main()
{
      Operation op4, op5, op6;

      op4 = Sqr;
      op5 = Cube;
      op6 = op4 + op5;
      op6(3); // 9 27
      Console.WriteLine(op6); // DelegateSample.Program+Operation
      Console.WriteLine(op6.GetType().BaseType); //
System.MulticastDelegate
      Console.WriteLine(op6.GetInvocationList().Length); // 2
      op4 = op6 - op4;
      op4(4); // 64
      op5 -= op5;
      op5(5); // NullReferenceException
      return 0;
}
```

4.8.2.3. Анонимные функции

Делегаты также могут создаваться с помощью анонимных функций, которые представляют собой «встроенные методы», создаваемые в процессе выполнения. Они объявляются внутри методов классов. Анонимные функции могут видеть локальные переменные окружающих их методов.

Синтаксис:

```
      <анонимная функция> :: <тип делегата> <идентификатор> = delegate
[([<список формальных параметров>])] блок;
      <анонимная функция> :: <тип делегата> <идентификатор> = <лямбда-
выражение>;
```

Обратите внимание, список формальных параметров и заключающие его скобки указывать не обязательно в том случае, если параметры при реализации анонимного метода не требуются. Однако при вызове делегата фактические параметры нужно указать обязательно.

Пример:

```
      delegate double Del1(double x);
      delegate double Del2();

      static int Main()
      {
            Del1 op7 = delegate(double x) { return -x; };
            Del1 op8 = new Del1(Math.Sin);
            Del1 op9 = delegate { return Math.PI; };
            Del2 op10 = delegate { return Math.PI; };
            Del1 op11 = (double x) => x * x;
            Del1 op12 = x => x * x;

            Console.WriteLine(op7(3)); // -3
            Console.WriteLine(op8(3)); // 0,141120008059867
            Console.WriteLine(op9(3)); // 3,14159265358979
            Console.WriteLine(op10()); // 3,14159265358979
            Console.WriteLine(op11(5)); // 25
            Console.WriteLine(op11(7)); // 49
            return 0;
      }
```

Лямбда-выражения в данном учебном пособии не рассматриваются.

4.8.3. Определение событий с помощью делегатов

Событие – это член, используемый классом или объектом для уведомления. Событие объявляется аналогично полю, однако оно должно иметь тип делегата и его объявление должно содержать модификатор **event**:

```
      <описание события> :: [<атрибуты>] [<модификаторы>] event <оператор
объявления>
```

```
      <описание события> :: [<атрибуты>] [<модификаторы>] event <тип>
<идентификатор> "{" <методы доступа> "}"

      <методы доступа> :: <метод добавления> <метод удаления>
      <методы доступа> :: <метод удаления> <метод добавления>

      <метод добавления> :: [<атрибуты>] add <блок>
      <метод удаления> :: [<атрибуты>] remove <блок>
```

Для события можно предусмотреть реакцию на добавление (метод **add**) и удаление (метод **remove**) обработчика. Причем либо эти методы вообще не указываются, либо указываются оба. Доступ к ссылке на добавляемый или удаляемый делегат обеспечивает **value**. Вызывать такое событие извне класса, как обычный делегат, нельзя.

В отличие от делегатов, вне класса, в котором объявлено событие, можно только добавить или удалить обработчик данного события (операциями «+=» и «–=»). Причем эти операции возвращают тип **void**, поэтому нет возможности получить доступ к базовому делегату события.

Как уже отмечалось, для события E типа делегата T зарезервированы сигнатуры «**void** add_E(T handler)» и «**void** remove_E(T handler)

Трудно увидеть пользу от механизма событий в консольном приложении Windows. Принцип работы консольного приложения не сильно отличается от работы программы в MS DOS – команды выполняются друг за другом, а все операции инициирует сам программист, поэтому нет необходимости уведомлять его о чем-либо. Все преимущества сообщений мы увидим, когда будем писать оконные приложения Windows.

Тем не менее, приведем небольшой пример:

```
      class Input
      {
            public delegate void InputDelegate (int num);
            private event InputDelegate OnInputNum;
            public event InputDelegate OnInput
            {
                  add /* реакция на добавление обработчика */
                  {
                        OnInputNum += value;
                        Console.WriteLine("Добавлен обработчик " +
value.GetType());
                  }
                  remove /* реакция на удаление обработчика */
                  {
                        OnInputNum -= value;
                        Console.WriteLine("Удален обработчик " +
value.GetType());
                  }
            }
            public event InputDelegate OnInputDiv2;
```

```
            public event InputDelegate OnInputDiv5;
            public event InputDelegate OnInputDiv10; // Ошибка
            void add_OnInputDiv10(InputDelegate handler) { }
            void remove_OnInputDiv10(InputDelegate handler) { }
            public void InputNumbers()
            {
                  string s;
                  int n;
                  do
                  {
                        Console.Write("Введите целое число (0 - стоп): ");
                        s = Console.ReadLine();
                        if (int.TryParse(s, out n))
                        {
                              OnInputNum(n);
                              if (n % 2 == 0) OnInputDiv2(n);
                              if (n % 5 == 0) OnInputDiv5(n);
                        }
                        else continue;
                  } while (n != 0);
            }
      }

      class Program
      {
            static int Main()
            {
                  Input i = new Input();
                  Input.InputDelegate d1 = delegate(int n) {
Console.WriteLine("Введено число 0x{0:X}", n); };
                  Input.InputDelegate d2 = delegate {
Console.WriteLine("Введено чётное число"); };
                  Input.InputDelegate d3 = delegate {
Console.WriteLine("Введено число, кратное 5"); };

                  i.OnInputDiv2(2); // Ошибка
                  i.OnInput += d1;
                  i.OnInputDiv2 += d2;
                  i.OnInputDiv5 += d3;
                  i.InputNumbers();
                  return 0;
            }
      }
```

Первая ошибка связана с тем, что требуемые для события сигнатуры уже зарезервированы в классе. Вторая – с попыткой вызвать событие как делегат извне класса.

 Пример: Samples\4.8\4_8_3_event.

Для событий, использующихся в системе классов .NET Framework, существуют рекомендации:

1) Базовый делегат события не должен возвращать значения.

2) Он должен принимать ровно два аргумента, первый из которых содержит ссылку на объект, инициировавший событие, а второй – на парамет-

ры события.

3) Параметры события должны быть представлены экземпляром класса EventArgs (для событий без параметров) или его потомка.

Для событий без параметров есть готовый делегат

```
public delegate void EventHandler(object sender, EventArgs args);
```

Это позволяет унифицировать систему событий и облегчить передачу сообщений между программами, написанными на разных языках программирования.

§ 4.9. Интерфейсы

Ключ к пониманию интерфейсов лежит в их сравнении с классами. Классы – это объекты, обладающие свойствами и методами, которые на эти свойства воздействуют. Хотя классы проявляют некоторые характеристики, связанные с поведением (методы), они представляют собой предметы, а не действия, присущие интерфейсам. Интерфейсы позволяют определять характеристики или возможности действий и применять их к классам независимо от иерархии последних.

Язык C# не поддерживает множественное наследование, так что невозможно произвести класс от нескольких базовых классов. А вот интерфейсы позволяют определять набор семантически связанных методов и свойств, которые способны реализовать избранные классы независимо от их иерархии.

Концептуально интерфейсы представляют собой связки между двумя в корне отличными частями кода. Иначе говоря, при наличии интерфейса и класса, определенного как реализующий (или наследующий) данный интерфейс, клиентам класса дается гарантия, что у класса реализованы все члены, определенные в интерфейсе.

В языке C# интерфейс – понятие первостепенной важности, объявляющее ссылочный тип, который включает только объявления членов. С точки зрения языка C++ можно воспринимать интерфейс как абстрактный класс, в котором объявлены только абстрактные методы. Наследуясь от такого класса, мы вынуждены перегрузить все объявленные в нем методы.

4.9.1. Объявление интерфейсов

Синтаксис описания интерфейса:

```
      [<атрибуты>] [<модификаторы>] interface <идентификатор> [<параметры
типа>] [: <список наследования>] [<ограничения параметров типа>]
      "{"
            [<тело интерфейса>]
      "}" [;]
```

Комментарии:

1) Атрибуты рассмотрены в § 5.4;

2) Допустимы модификаторы **new**, **public**, **protected**, **internal**, **private** и **partial**. По смыслу они аналогичны таковым для классов;

3) Идентификатор может быть любым, но принято начинать его с за-

главной буквы «I» (например, ICloneable, IMyInterface и т.п.);

4) Параметры типа и ограничения параметров типа будут рассмотрены в § 5.1;

5) В списке наследования могут быть только другие интерфейсы. Уровень доступа явных базовых интерфейсов должен быть не ниже уровня доступа самого интерфейса;

6) В теле интерфейса могут содержаться объявления методов, свойств, индексаторов и событий. Формат описания для этих сущностей практически такой же, как в классе, но ни одна из них не реализуется в самом интерфейсе. Т.е. подразумевается модификатор **abstract**, хотя явно он не указывается. В интерфейсах вообще запрещены модификаторы членов, кроме **new** при перекрытии старого члена новым и **event** при описании событий. В остальном синтаксис описания членов соответствует синтаксису описания абстрактных членов класса. Т.е. тела методов и событий отсутствуют, а у свойств и индексаторов отсутствуют тела методов доступа.

Пример:

```
public delegate void TestD();

public interface ITest
{
      void F(out string s); // OK
      int Count { get; } // OK
      string this[int i] { get; set; } // OK
      string this[int i, int j]; // Ошибка
      public abstract int X(); // Ошибка
      event TestD TestEvent1; // OK
      event TestD TestEvent2 { add; remove; } // Ошибка
}
```

 Пример: Samples\4.9\4_9_1_decl.

4.9.2. Реализация интерфейсов

Если класс реализует интерфейс, то он должен определить все члены, описанные в нем. Иначе возникает ошибка компиляции. Также все неявные члены интерфейса должны быть реализованы в классе с модификатором **public**.

Пример:

```
public delegate void TestD();

public interface ITest
{
```

```
        void F(out string s);
        int Count { get; }
        string this[int i] { get; set; }
        event TestD TestEvent;
    }

    class MyClass1 : ITest // Ошибка
    {
    }

    class MyClass2 : ITest // OK
    {
        public void F(out string s)
        {
            s = "MyClass2";
        }

        public int Count
        {
            get { return 2; }
        }

        public string this[int i]
        {
            get { return "MyClass2.ITest." + i; }
            set { }
        }

        public event TestD TestEvent;
    }
```

 Пример: Samples\4.9\4_9_2_impl.

4.9.2.1. Явная квалификация имени члена интерфейса

Мы употребили термин «неявные члены интерфейса». Что он означает? При описании различных членов класса мы говорили о том, что имя интерфейса может быть в реализующем классе указано явно:

```
<тип интерфейса>.<имя члена>
```

Такой вариант синтаксиса называется *явной квалификацией* имени члена интерфейса. Если имя не указано, квалификация неявная. Зачем может потребоваться явная квалификация?

Во-первых, для сокрытия имени с помощью интерфейсов. При явной реализации интерфейса запрещено использование модификаторов доступа, т.е. они имеют модификатор доступа по умолчанию (**private**). Поскольку к явным реализациям члена интерфейса нельзя получить доступ через экземпляры классов или структур, они позволяют исключить реализации интерфейса из списка доступных членов класса или структуры. Это особенно по-

лезно в том случае, если в классе или структуре реализуется внутренний интерфейс, который не предоставляет интереса для объекта, где используется этот класс или эта структура. Пример:

```
public interface ITest2
{
      int Count();
}

class MyClass3 : ITest2
{
      public int ITest2.Count() { return 3; } // Ошибка
      int ITest2.Count() { return 3; } // OK
}

class MyClass4 : ITest2
{
      public int Count() { return 4; } // OK
}

static int Main()
{
      MyClass3 c3 = new MyClass3();
      MyClass4 c4 = new MyClass4();

      Console.WriteLine(c3.Count()); // Ошибка
      Console.WriteLine(c4.Count()); // 4
      return 0;
}
```

Во-вторых, для избежания неоднозначности имен. Явные реализации члена интерфейса позволяют разрешить неоднозначности членов интерфейса с одинаковой сигнатурой. Без этого было бы невозможно иметь разные реализации члена интерфейса с одинаковыми сигнатурой и типом возвращаемого значения, а также было бы невозможно включить в класс или структуру любые реализации членов интерфейса с одинаковой сигнатурой и разными типами возвращаемых значений. Пример:

```
class MyClass5 : ITest, ITest2
{
      public void F(out string s)
      {
            s = "MyClass5";
      }

      int ITest.Count
      {
            get { return 5; }
      }

      public string this[int i]
      {
            get { return "MyClass5.ITest." + i; }
            set { }
```

```
        }

        int ITest2.Count()
        {
              return 5;
        }

        public event TestD TestEvent;
  }
```

В данном примере без явной квалификации наблюдался бы конфликт имен для членов Count.

Доступ к явной реализации члена интерфейса можно получить, используя преобразование ссылки на экземпляр класса к типу интерфейса:

```
  static int Main()
  {
        MyClass3 c3 = new MyClass3();

        Console.WriteLine(((ITest2)c3).Count()); // 3
        return 0;
  }
```

4.9.2.2. Запрос о реализации интерфейса

Что будет, если клиент попытается использовать класс так, как если бы в нем был реализован метод, на самом деле в нем не реализованный? Проверим:

```
  MyClass2 c2 = new MyClass2();

  Console.WriteLine(((ITest2)c2).Count());
```

Получаем исключительную ситуацию System.InvalidCastException. Поэтому при получении ссылки на объект и не будучи уверенными в том, что для данного объекта реализуется тот или иной интерфейс, это нужно предварительно проверить с помощью оператора **is** или **as**:

```
  static void F1(object x)
  {
        if (x is ITest2) Console.WriteLine(((ITest2)x).Count());
        else Console.WriteLine("Интерфейс ITest2 не реализован");
  }

  static void F2(object x)
  {
        ITest2 intf = x as ITest2;
        if (intf != null) Console.WriteLine(intf.Count());
        else Console.WriteLine("Интерфейс ITest2 не реализован");
  }

  static int Main()
  {
```

```
        MyClass2 c2 = new MyClass2();
        MyClass4 c4 = new MyClass4();

        F1(c2); // Интерфейс ITest2 не реализован
        F2(c2); // Интерфейс ITest2 не реализован
        F2(c4); // 4
        return 0;
    }
```

4.9.2.3. Доступ к членам интерфейса

Любой класс, реализующий интерфейсы, имеет неявное преобразование типа к типам этих интерфейсов. Соответственно, доступ к реализованным членам интерфейса осуществляется так же, как доступ к членам класса с такой же сигнатурой. При этом доступ возможен и по ссылке на экземпляр класса, и по ссылке на реализацию интерфейса.

Пример:

```
    static void F3(ITest i)
    {
        string s;
        i.F(out s);
        Console.WriteLine(s);
        Console.WriteLine(i.Count);
        Console.WriteLine(i[5]);
    }

    static int Main()
    {
        MyClass2 c2 = new MyClass2();

        F3(c2);
        return 0;
    }
```

Вывод на консоль:

```
    MyClass2
    2
    MyClass2.ITest.5
```

Получить ссылку на интерфейс экземпляра класса MyClass5 не получится, т.к. он реализует сразу два интерфейса. Однако, можно эти интерфейсы комбинировать:

```
    interface ITestComb : ITest, ITest2
    {
    }

    class MyClass5 : ITestComb
    // и т.д.
```

А чтобы не возникала неоднозначность при доступе к члену Count, ис-

пользуем преобразование типа ссылки на реализацию интерфейса:

```
static void F4(ITestComb i)
{
      string s;
      i.F(out s);
      Console.WriteLine(s);
      Console.WriteLine(((ITest)i).Count);
      Console.WriteLine(((ITest2)i).Count());
      Console.WriteLine(i[7]);
}

static int Main()
{
      MyClass5 c5 = new MyClass5();

      F4(c5);
      return 0;
}
```

Вывод на консоль:

```
MyClass5
5
5
MyClass5.ITest.7
```

4.9.3. Интерфейсы и наследование

Рассмотрим особенности реализации интерфейсов при наследовании классов, их реализующих.

 Пример: Samples\4.9\4_9_3_inherit.

4.9.3.1. Наследование реализаций интерфейсов

Класс наследует все реализации интерфейсов, содержащиеся в его базовых классах. Без явной повторной реализации интерфейса, в производном классе нельзя изменить сопоставление интерфейсов, унаследованных им из базовых классов. Тем не менее, при сопоставлении метода интерфейса с виртуальным методом в классе можно переопределить этот виртуальный метод в производных классах и изменить реализацию интерфейса. Пример:

```
interface ITest
{
      void Method1();
      void Method2();
}

class Base : ITest
{
      public void Method1()
```

```
        {
            Console.WriteLine("Base.Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base.Method2");
        }
    }

    class Child : Base
    {
        public new void Method1()
        {
            Console.WriteLine("Child.Method1");
        }

        public override void Method2()
        {
            Console.WriteLine("Child.Method2");
        }
    }

    static int Main()
    {
        Base b = new Base();
        Child c = new Child();

        b.Method1(); // Base.Method1
        c.Method1(); // Child.Method1
        ((ITest)b).Method1(); // Base.Method1
        ((ITest)c).Method1(); // Base.Method1

        b.Method2(); // Base.Method2
        c.Method2(); // Child.Method2
        ((ITest)b).Method2(); // Base.Method2
        ((ITest)c).Method2(); // Child.Method2
        return 0;
    }
```

Так как явные реализации метода интерфейса не могут быть объявлены виртуальными, переопределить явную реализацию члена интерфейса невозможно. Однако в явной реализации члена интерфейса можно вызвать другой метод, который может быть объявлен виртуальным, что позволяет переопределить его в производных классах.

4.9.3.2. Повторная реализация интерфейсов

В классе, который наследует реализацию интерфейса, разрешается повторно реализовать этот интерфейс путем включения его в список базовых классов. В процедуре сопоставления повторно реализованных интерфейсов участвуют унаследованные объявления открытых членов и унаследованные явные объявления членов интерфейса. Пример:

```
interface ITest2
{
      string A();
      string B();
      string C();
      string D();
}

class Base2 : ITest2
{
      public string A() { return "B.A"; }
      public string B() { return "B.B"; }
      string ITest2.C() { return "B.C"; }
      string ITest2.D() { return "B.D"; }
}

class Child2 : Base2, ITest2
{
      string ITest2.A() { return "C.A"; }
      public string D() { return "C.D"; }
}

static int Main()
{
      Child2 c2 = new Child2();

      Console.WriteLine(c2.A()); // B.A
      Console.WriteLine(c2.B()); // B.B
      Console.WriteLine(((ITest2)c2).C()); // B.C
      Console.WriteLine(((ITest2)c2).A()); // C.A
      Console.WriteLine(c2.D()); // C.D
      return 0;
}
```

Вывести на консоль строку «B.D», имея экземпляр дочернего класса, не получится, т.к. соответствующая реализация является явной, а при преобразовании к ссылке на реализацию интерфейса будет вызываться метод D() дочернего класса.

4.9.3.3. Абстрактные классы и интерфейсы

В абстрактных классах, как и в классах, не являющихся абстрактными, необходимо указать реализацию всех членов интерфейсов, включенных в список их базовых классов. Тем не менее, в абстрактном классе разрешается сопоставлять методы интерфейса абстрактным методам. Пример:

```
public delegate void TestD();

interface ITest3
{
      event TestD TestEvent;
}

abstract class AbsClass : ITest2
```

```
    {
        public abstract string A();
        public string B() { return ""; }
        public abstract string C();
        string ITest2.D() { return ""; }
        public abstract event TestD TestEvent;
    }
```

4.9.4. Примеры использования интерфейсов

Многие классы, с которыми мы познакомились к настоящему времени, реализуют те или иные интерфейсы. Кроме того, в некоторых ситуациях бывает полезно реализовать в создаваемом классе определенные интерфейсы – это может расширить список операций, доступных для обработки данных.

Как мы помним, структуры и классы типов данных реализуют следующие интерфейсы:

• Типы по значению (кроме структур) – IComparable, IFormattable (кроме логического и символьного типа), IConvertible, IEquatable (кроме перечислений);

• Структура DateTime – IComparable, IFormattable, IConvertible;

• Класс String – IComparable, ICloneable, IConvertible, IEnumerable;

• Класс Array – ICloneable, IList, ICollection, IEnumerable.

Еще мы говорили об интерфейсе IDisposable, использующемся в операторе выделения ресурсов **using**. Что это им дает?

 Пример: Samples\4.9\4_9_4_samples.

4.9.4.1. Ранжирование объектов

Интерфейс IComparable описывает всего один член:

```
    int CompareTo(object obj);
```

Реализация этого метода должна возвращать –1, если логика приложения определяет, что текущий экземпляр объекта предшествует экземпляру obj, +1, если текущий экземпляр следует за obj, и 0, если их ранг совпадает. Для чисел, дат, строк и т.п. можно провести соответствие реализации данного интерфейса и операций отношения (табл. 4.4).

Табл. 4.4 – Операции отношения и метод CompareTo

Операция отношения	Результат метода CompareTo
Равно (`x == y`)	`x.CompateTo(y) == 0`

Не равно (x != y)	x.CompateTo(y) != 0
Больше (x > y)	x.CompateTo(y) > 0
Больше или равно (x >= y)	x.CompateTo(y) >= 0
Меньше (x < y)	x.CompateTo(y) < 0
Меньше или равно (x <= y)	x.CompateTo(y) <= 0

Чаще всего экземпляры классов, реализующих данный интерфейс, используются при ранжировании объектов, сортировке массивов данных и поиске. Например, массив экземпляров любого класса, реализующего этот интерфейс, можно сортировать при помощи методов Array.Sort, либо осуществлять в нем двоичный поиск вызовом Array.BinarySearch.

Пример:

```
class Rational : IComparable
{
      public int Numerator = 0;
      public int Denominator = 1;

      public Rational (int num, int den)
      {
            Numerator = num;
            Denominator = den;
      }

      public int CompareTo(object obj)
      {
            if (obj is Rational)
            {
                  Rational r = (Rational)obj;
                  return Math.Sign((double)Numerator / Denominator -
                        (double)r.Numerator / r.Denominator);
            }

            throw new ArgumentException("Нельзя дробь сравнить с " +
                  (obj != null ? obj.GetType().ToString() : "null"));
      }

      public override string ToString()
      {
            return String.Format("{0}/{1}", Numerator, Denominator);
      }
}

static int Main()
{
      Rational[] r = new Rational[3];

      r[0] = new Rational(11, 5);
      r[1] = new Rational(-1, 6);
      r[2] = new Rational(3, 8);

      Array.Sort(r);
      foreach (Rational x in r) Console.WriteLine(x);
```

```
            return 0;
      }
```

Данный класс позволяет сортировать рациональные дроби. Вывод на консоль:

```
      -1/6
      3/8
      11/5
```

Если не реализовать в классе Rational интерфейс IComparable, при выполнении получим исключение System.InvalidOperationException.

Более простой вариант ранжирования обеспечивает обобщенный интерфейс IEquatable<T>. Он также описывает только один член:

```
      bool Equals(T other);
```

Реализация данного метода позволяет определить равенство или неравенство двух экземпляров объектов. Классам, реализующим данный интерфейс, можно осуществлять поиск, удаление и другие операции в универсальных коллекциях типа Dictionary<TKey, TValue>, List<T> и т.п., некоторые перегрузки метода Array.BinarySearch и т.д.

Пример:

```
      class Rational : IEquatable<Rational>
      {
            /* описанное ранее тело класса */

            public bool Equals(Rational r)
            {
                  return (long)Numerator * r.Denominator ==
                        (long)r.Numerator * Denominator;
            }
      }

      static int Main()
      {
            /* описанное ранее использование класса */

            List<Rational> list = new List<Rational>();

            list.AddRange(r);
            Console.WriteLine("Индекс элемента 11/5: " + list.IndexOf(new
Rational(11, 5))); // 2
            return 0;
      }
```

4.9.4.2. Перечисления и коллекции

Перечислители

Основное назначение интерфейса IEnumerable – предоставление перечислителя, который поддерживает простой перебор элементов коллекции (итератор). Его реализации чаще всего используются в цикле **foreach**.

Интерфейс IEnumerable содержит объявление единственного метода:

```
IEnumerator GetEnumerator();
```

Этот метод возвращает экземпляр класса, реализующего другой интерфейс – перечислитель IEnumerator. Оба этих интерфейса объявлены в пространстве имен System.Collections. Члены интерфейса IEnumerator:

```
void Reset();
bool MoveNext();
object Current;
```

Первый метод устанавливает перечислитель в начальное положение. Второй осуществляет переход к следующему элементу перечисления. Если он возвращает **false**, это означает, что больше элементов нет. И последнее свойство возвращает текущий элемент. В этом случае оператор

```
foreach (<тип> v in x) <внедряемый оператор>
```

развертывается следующим образом:

```
{
      IEnumerator e = x.GetEnumerator();
      try
      {
            <тип> v;
            e.Reset();
            while (e.MoveNext())
            {
                  v = (<тип>)e.Current;
                  <внедряемый оператор>
            }
      }
      finally
      {
            // Удаление ресурсов
      }
}
```

Пример:

```
class VowelEnum : IEnumerator
{
      private string S;
      private int pos;
```

```
        public VowelEnum(string s) { S = s; }

        public void Reset()
        {
            pos = -1;
        }

        public bool MoveNext()
        {
            pos = S.IndexOfAny("аеёиоуыэюя".ToCharArray(), pos + 1);
            return pos >= 0 && pos < S.Length;
        }

        public object Current
        {
            get { return S[pos]; }
        }
    }

    class Word : IEnumerable
    {
        public string S = string.Empty;
        public Word(string s) { S = s; }

        public IEnumerator GetEnumerator()
        {
            return new VowelEnum(S);
        }
    }

    static int Main()
    {
        string str = "Будем искать в этой фразе гласные буквы!";
        Word w = new Word(str);

        Console.Write("Гласные: ");
        foreach (char c in w) Console.Write("{0} ", c);
        Console.WriteLine();
        return 0;
    }
```

В этом примере в качестве итератора выступает класс Word, инкапсулирующий строку, а в качестве перечислителя – класс VowelEnum, перебирающий в строке все гласные буквы. Вывод на консоль:

```
Гласные: у е и а э о а е а ы е у ы
```

Управляемые итераторы

Как уже было сказано в п. 3.4.4.6, ключевое слово **yield** может быть использовано для создания управляемого итератора. При этом конструкция **yield return** возвращает следующий элемент итератора, а **yield break** сообщает о конце итераций.

Пример:

```
public static IEnumerable VowelIterator(string s)
{
      int pos = -1;
      do
      {
            pos = s.IndexOfAny("аеёиоуыэюя".ToCharArray(), pos + 1);
            if (pos >= 0 && pos < s.Length) yield return s[pos];
            else yield break;
      } while (true);
}

static int Main()
{
      string str = "Будем искать в этой фразе гласные буквы!";

      Console.Write("Ещё раз: ");
      foreach (char c in VowelIterator(str)) Console.Write("{0} ", c);
      Console.WriteLine();
      return 0;
}
```

Эффект от этого кода такой же, как и в предыдущем примере:

```
Ещё раз: у е и а э о а е а ы е у ы
```

Коллекции

Интерфейс ICollection определяет размер, перечислители и методы синхронизации для всех нестандартных коллекций. Он наследует интерфейс IEnumerable. Интерфейс IList представляет коллекцию объектов с индивидуальным доступом, осуществляемым при помощи индекса. Он наследует интерфейсы ICollection и IEnumerable. Все эти интерфейсы объявлены в пространстве имен System.Collections.

Также есть интерфейс универсальной коллекции IList<T>, объявленный в пространстве имен System.Collections.Generic. Об универсальных типах мы будем говорить в следующей главе.

4.9.4.3. Копирование объектов

Для создания новой копии объекта, а не копирования ссылки в операции присвоения, используется реализация интерфейса ICloneable. В нем определен единственный метод:

```
object Clone();
```

Для примера обеспечим копирование дробей:

```
class Rational : ICloneable
{
      /* описанное ранее тело класса */
```

```
        public object Clone()
        {
            return new Rational(Numerator, Denominator);
        }

        public Rational Copy()
        {
            return (Rational)Clone();
        }
    }

    static int Main()
    {
        /* описанное ранее использование класса */

        r[1] = r[0];
        r[2] = (Rational)r[0].Clone();
        r[0].Numerator = 7;
        Console.WriteLine(r[1]); // 7/6
        Console.WriteLine(r[2]); // -1/6
        return 0;
    }
```

Экземпляр r[1] является ссылкой на r[0], а r[2] – копией. Чтобы не требовалось преобразование типа при копировании, можно использовать метод Copy.

4.9.4.4. Форматирование объектов

Интерфейс IConvertible определяет методы, которые преобразуют значение реализующего этот метод ссылочного типа или типа значения в другой тип данных. В этом интерфейсе описаны следующие методы:

```
TypeCode GetTypeCode();
<тип> To<класс>(IFormatProvider provider);
object ToType(Type conversionType, IFormatProvider provider);
```

Первый возвращает код типа объекта (см. п. 3.1.2), а остальные методы обеспечивают преобразование к ряду типов.

Пример:

```
class Rational : IConvertible
{
    /* описанное ранее тело класса */

    public TypeCode GetTypeCode()
    {
        return TypeCode.Object;
    }

    public bool ToBoolean(IFormatProvider provider)
    {
        return Numerator != 0 && Denominator != 0;
```

```
            }

            public byte ToByte(IFormatProvider provider)
            {
                  return (byte)(Numerator / Denominator);
            }

            public char ToChar(IFormatProvider provider)
            {
                  throw new InvalidCastException("Ошибка преобразования
Rational в char");
            }

            public double ToDouble(IFormatProvider provider)
            {
                  return (double)Numerator / Denominator;
            }

            public string ToString(IFormatProvider provider)
            {
                  return ToString();
            }
      }

      static int Main()
      {
            /* описанное ранее использование класса */

            Console.WriteLine(r[0].ToSingle(null)); // 1,166667
            return 0;
      }
```

Здесь перечислены не все методы ввиду их большого количества. При преобразовании к булевому типу значение дроби будет считаться истиной, если ее числитель и знаменатель не равны нулю. Для целых типов данных делим числитель на знаменатель, преобразуем тип результата и возвращаем его. Для типов с плавающей точкой сначала преобразуем числитель к требуемому типу, и только потом делим, чтобы деление не было целочисленным. Преобразование к строке мы уже определили ранее в методе ToString(). Если формат не поддерживается, генерируем исключительную ситуацию InvalidCastException. Интерфейс IFormatProvider в данном случае не используется, однако, если преобразование зависит от каких-то дополнительных опций (формат значения, региональные параметры и т.п.), можно указать экземпляр класса, его реализующий.

Интерфейс IFormattable используется для преобразования произвольного типа в строку, и определяет только один метод:

```
      string ToString(string format, IFormatProvider formatProvider);
```

Пример:

```
      class Rational : IFormattable
      {
            /* описанное ранее тело класса */

            public string ToString(string format, IFormatProvider
formatProvider)
            {
                  if (format == null || format[0] == 'G') return
                        Numerator.ToString(format, formatProvider) + "/" +
                        Denominator.ToString(format, formatProvider);
                  else if (format[0] == 'F') return
                        ToDouble(null).ToString(format, formatProvider);
                  else if (format[0] == 'D') return
                        ToInt32(null).ToString(format, formatProvider);
                  else throw new FormatException("Недопустимый формат");
      }

      static int Main()
      {
            /* описанное ранее использование класса */

            Console.WriteLine(String.Format("{0:F}", r[0])); // 1,17
            Console.WriteLine(String.Format("{0:F4}", r[0])); // 1,1667
            Console.WriteLine(r[0].ToString("F", new CultureInfo("en-US")));
// 1.17
            Console.WriteLine(String.Format("{0:D}", r[0])); // 1
            return 0;
      }
```

Интерфейс IFormatProvider предоставляет механизм извлечения объекта для управления форматированием. Как уже отмечалось (см. п. 3.2.1.1), его реализуют три класса .NET – NumberFormatInfo, DateTimeFormatInfo и CultureInfo. В принципе, нет необходимости создавать новые реализации данного интерфейса. Однако, если это потребуется, нужно будет реализовать лишь один его метод:

```
      object GetFormat(Type formatType);
```

Дополнительную информацию можно найти в библиотеке MSDN.

4.9.4.5. Управление ресурсами

В языке C# предусмотрено использование оператора **using** для описания блока управления ресурсами (см. п. 3.5.3). При этом управляемые ресурсы должны наследовать интерфейс IDisposable. Учитывая, что в среде CLR удалением неиспользуемых ресурсов занимается сборщик мусора, задача ручного управления освобождения ресурсов возникает не так уж часто.

Обычно это требуется в том случае, если при выполнении какой-либо операции объект использует неуправляемые ресурсы, особенно в критических объемах (большой объем памяти, множество потоков ввода-вывода и

т.п.). В этом случае, после завершения операции ресурсы можно освободить, но когда сборщик мусора сделает это – неизвестно.

Обычно код выглядит так:

```
public class MyResource: IDisposable
{
	private bool disposed;

	public MyResource()
	{
		// Выделяем ресурсы
		disposed = false;
	}

	public void Dispose()
	{
		Dispose(true);
		GC.SuppressFinalize(this);
	}

	private void Dispose(bool managed)
	{
		if (!disposed)
		{
			if (managed)
			{
				// Освобождаем управляемые ресурсы
			}

			// Освобождаем неуправляемые ресурсы
			disposed = true;
		}
	}

	~MyResource()
	{
		Dispose(false);
	}
}
```

Метод GC.SuppressFinalize запрещает сборщику мусора удалять управляемые ресурсы, ассоциированные с данным экземпляром объекта. При вызове метода Dispose мы их освободим сами, вместе с неуправляемыми ресурсами. Деструктором удаляются только неуправляемые ресурсы, т.е. удаление управляемых, если они не были освобождены, поручается сборщику мусора. Теперь можно использовать экземпляры данного класса в блоках управления ресурсами:

```
// Неизвестно, когда ресурсы будут освобождены
MyResource res1 = new MyResource();
// Ресурсы будут освобождены после выхода из блока using
using (MyResource res2 = new MyResource()) { }
```

5. СПЕЦИАЛЬНЫЕ ВОЗМОЖНОСТИ

Рассмотрим кратко некоторые дополнительные возможности, предоставляемые .NET Framework.

§ 5.1. Универсальные типы

Универсальные члены (или универсальные шаблоны, generic members) – это некоторый аналог механизма шаблонов языка C++. Универсальными членами могут являться классы, структуры, интерфейсы, делегаты (*универсальные типы*), а также методы, свойства, индексаторы (*универсальные методы*), которые имеют прототипы (параметры типов) для одного или нескольких типов, которые они хранят или используют. Класс универсальной коллекции может использовать параметр типа в качестве типа объектов, которые в нем хранятся. Параметры типа отображаются как типы его полей и типы параметров его методов. Универсальный метод может использовать параметр типа как тип своего возвращаемого значения или как тип одного из своих формальных параметров.

В противоположность универсальным, будем называть не имеющие параметров типа члены *конкретными членами*.

5.1.1. Параметры типа

При описании синтаксиса типов и их членов мы использовали обозначения «параметры типа» и «ограничения параметров типа». Разберемся сначала с параметрами типа:

```
<параметры типа> :: "<" <формальный список типов> ">"
<формальный список типов> :: <идентификатор> [,...]
```

Таким образом, указание параметров типа заключается в перечислении в угловых скобках одного или более идентификаторов, разделенных запятой. Эти идентификаторы являются формальным списком типов. Для типов с однобуквенными параметрами рекомендуется использовать «T» в качестве имени параметра типа (если должно быть несколько однобуквенных параметров, можно использовать следующие буквы алфавита – «U», «V», …). К описательным именам параметров типа рекомендуется добавлять префикс «T» (например, TElement, TValue и т.д.).

После этого в данном универсальном типе или методе можно исполь-

зовать эти идентификаторы в качестве типов данных. Если тип является универсальным, то его параметры типа могут использовать все члены объекта, в том числе вложенные типы.

 Пример: Samples\5.1\5_1_1_generic.

5.1.1.1. Использование универсальных типов

Если мы описали универсальный тип, то теперь его имя обязательно должно включать фактический список типов:

```
<имя универсального типа> :: <имя типа> "<"<фактический список типов>">"
<фактический список типов> :: <тип> [, ...]
```

В качестве фактических типов могут выступать стандартные типы данных, ранее объявленные типы (делегаты, перечисления, классы, структуры и т.д.) или, если тип вложен в другой универсальный тип, то можно использовать параметры вмещающего универсального типа. Если фактические типы, соответствующие всем формальным типам, не указаны, или указаны неизвестные типы, возникает ошибка компиляции.

Если мы хотим использовать универсальный метод (универсального или обычного типа), то синтаксис будет следующим:

```
<вызов статического универсального метода> ::
      <имя универсального типа>.<вызов универсального метода>
      <имя конкретного типа>.<вызов универсального метода>

<вызов универсального метода экземпляра> ::
      <имя экземпляра>.<вызов универсального метода>

<вызов универсального метода> :: <имя метода> ["<" <фактический список
типов> ">"] ([<фактические параметры вызова>]);
```

Фактический список типов метода указывать не обязательно, если его можно определить по типам фактических параметров вызова.

Пример:

```
class MyClass1<int> { } // Ошибка
class MyClass2<T, U<V>> { } // Ошибка

class MyClass<T>
{
      private T field;
      public delegate void NotifyEvent(MyClass<T> sender);
      public event NotifyEvent OnChange;

      public MyClass(T value)
      {
            field = value;
```

```
            }

            public T Prop
            {
                  get { return field; }
                  set
                  {
                        field = value;
                        if (OnChange != null) OnChange(this);
                  }
            }

            public override string ToString()
            {
                  return field.ToString();
            }

            public bool EqualTo<U>(U value)
            {
                  return field.Equals(value);
            }

            public static bool EqualTo<U>(T value1, U value2)
            {
                  return value1.Equals(value2);
            }
      }

      class Program
      {
            static void Out<T1, T2>(T1 var1, T2 var2)
            {
                  Console.WriteLine("{0} & {1}", var1.GetType(),
var2.GetType());
            }

            static int Main()
            {
                  MyClass<int> a = new MyClass<int>(5);
                  Console.WriteLine(a); // 5
                  Console.WriteLine(a.EqualTo(5)); // True
                  Console.WriteLine(MyClass.EqualTo(1, 1)); // Ошибка
                  Console.WriteLine(MyClass<int>.EqualTo(1, 1.0)); // False
                  Console.WriteLine(a.EqualTo<float>(5)); // False
                  Console.WriteLine();

                  MyClass<string> s = new MyClass<string>("!!!");
                  MyClass<MyClass<string>> s2 = new
MyClass<MyClass<string>>(s);
                  Console.WriteLine(s2); // !!!

                  MyClass<string>.NotifyEvent n = delegate(MyClass<string>
cls) {
                        Console.WriteLine("Значение изменилось на " +
cls.Prop);
                  };

                  a.OnChange += n; // Ошибка
                  s.OnChange += n;
                  s.Prop = "???"; // Значение изменилось на ???
```

```
                    Console.WriteLine();

                    Out(1, 1.5); // System.Int32 & System.Double
                    Out<double, double>(1, 1.5); // System.Double &
System.Double

                    return 0;
              }
        }
```

Ошибки:

1) **int** является типом, а не идентификатором;

2) U<V> не является идентификатором;

3) Имя универсального класса использовано без фактического списка параметров типа;

4) MyClass<string>.NotifyEvent и MyClass<int>.NotifyEvent – разные типы делегатов.

Как видно, у класса фактические параметры типа должны быть указаны всегда, а у методов – нет, если аргументы этих типов используются в списке параметров вызова. В этом случае происходит процесс, называемый *выводом типов*. Например, при первом вызове метода Out параметры имеют типы **int** и **double**. Если нас устраивает, что T1 = **int** и T2 = **double**, то фактические параметры типа указывать не обязательно. Во втором случае мы указали их явно.

5.1.1.2. Отличие от шаблонов языка C++

По сравнению с шаблонами языка C++, универсальные члены имеют ряд ограничений. В частности:

• Нельзя использовать параметр типа в качестве базового класса:

```
        class MyClass3<T> : T { } // Ошибка
```

• Нельзя использовать не являющиеся типами параметры шаблона:

```
        class MyClass4<int N> { } // Ошибка
```

Но самое серьезное отличие состоит в том, что язык C++ позволяет использование кода, который может быть допустимым не для всех параметров типа в шаблоне, и который затем проверяется для конкретного типа, используемого в качестве параметра. Например:

```
        template <class T> class MyClass
        {
        public:
              MyClass(T value)
```

```
        {
            value.X();
        }
};
```

Этот код компилятор C++ воспринимает как корректный. Ошибка может возникнуть позже, когда мы укажем в качестве параметра тип, не имеющий метода X().

В языке C# код класса должен быть изначально написан таким образом, чтобы он работал с любым типом. Если не указаны ограничения параметров типа (см. п. 5.1.2), то очевидно, что к параметрам типа можно применять только те операции, которые применимы ко всем типам .NET. Например:

```
class MyClass5<T>
{
    T value = 0; // Ошибка
    public MyClass5(T b)
    {
        T a;
        a = b; // OK
        if (a == b) // Ошибка
        {
            a = null; // Ошибка
            if (a == null) // OK
            {
                b.GetType(); // OK
            }
        }
    }
}
```

Т.к. не все типы допускают неявное преобразование из типа **int**, компилятор C# сразу выдает ошибку для «value = 0», хотя, вполне возможно, мы бы использовали этот класс только для числовых параметров типа. Операция «a = b» компилируется без ошибки, т.к. присвоение поддерживают все типы данных. Операция сравнения «a == b» не компилируется, т.к. не все типы ее поддерживают (например, структуры). Операция присваивания «a = **null**» также вызывает ошибку, т.к. пустой ссылкой можно инициализировать только обнуляемые и ссылочные типы, а типы по значению – нет. Операция сравнения «a == **null**» разрешена. Сравнить с пустой ссылкой можно экземпляр любого типа, просто для типа по значению результатом всегда будет **false**. Ну и метод GetType() определен для любого типа, т.к. наследуется от корневого класса Object.

5.1.1.3. Инициализация значениями по умолчанию

В универсальных типах и методах одной из существующих проблем является назначение параметризованному типу T значения по умолчанию, если заранее неизвестны следующие моменты:

• Является ли T ссылочным типом или типом значения;

• Если T является типом значения, будет ли он числовым значением или структурой.

Очевидно, что для всех этих трех случаев инициализация существенно отличается. Для ссылочного типа значением по умолчанию является **null**, для числовых типов – аналог нуля для соответствующего типа данных (см. § 3.1), для структур необходимо все поля инициализировать значениями по умолчанию. Например, напишем класс-вектор с произвольным типом аргументов:

```
class Vector<T>
{
	private T[] elems;

	public Vector(int size)
	{
		elems = new T[size];
	}

	public void Reset()
	{
		for (int i = 0; i < elems.Length; i++)
		{
			// elems[i] = ???;
		}
	}

	public T this[int i]
	{
		get { return elems[i]; }
		set { elems[i] = value; }
	}

	public int Length
	{
		get { return elems.Length; }
	}
}
```

По замыслу, метод Reset должен сбрасывать значения элементов вектора. Но что написать справа от оператора присваивания? На значение любого типа компилятор будет выдавать ошибку. Нет такого типа данных, который имеет явное или неявное преобразование ко всем другим типам.

Для этого используется ключевое слово **default**. О нем мы уже говори-

ли в § 3.1. Выражение **default**(<тип>) возвращает значение по умолчанию для указанного типа данных:

```
elems[i] = default(T);
```

5.1.1.4. Наследование от универсальных классов и интерфейсов

Введем понятие *открытого типа* для универсальных типов, имеющих формальные параметры (например, Vector<T>). После указания фактических параметров получаем *закрытый тип* (например, Vector<**int**>).

При наследовании от универсального типа мы можем получить либо универсальный тип, используя наследование открытого типа, либо конкретный тип при наследовании закрытого типа. Впрочем, производный тип может объявлять свои параметры типа, не зависящие от базового типа. Пример:

```
class BaseClass { }
class BaseGenClass<T> { }

class ChildClass1 : BaseClass { } // OK
class ChildClass2 : BaseGenClass<int> { } // OK
class ChildClass3 : BaseGenClass<T> { } // Ошибка

class ChildGenClass1<T> : BaseClass { } // OK
class ChildGenClass2<T> : BaseGenClass<T> { } // OK
class ChildGenClass3<U> : BaseGenClass<T> { } // Ошибка
class ChildGenClass4<U> : BaseGenClass<int> { } // OK
class ChildGenClass5<T, U> : BaseGenClass<T> { } // OK

class ChildSubClass<T> : ChildGenClass5<T, int> { } // OK
class ChildSubClass : ChildGenClass5<double, int> { } // OK
```

Открытые и закрытые типы можно использовать в качестве параметров методов:

```
static void Swap<T>(Vector<T> a, Vector<T> b, int idx)
{
      T x = a[idx];
      a[idx] = b[idx];
      b[idx] = x;
}

static void Swap(Vector<int> a, Vector<double> b, int idx)
{
      int x = a[idx];
      a[idx] = (int)b[idx];
      b[idx] = x;
}
```

Если универсальный класс или структура реализует интерфейс, все экземпляры этого класса или структуры могут быть приведены к этому интерфейсу:

```
class CloneClass<T> : ICloneable
{
      public object Clone()
      {
            return new CloneClass<T>();
      }
}

static object CreateCopy(ICloneable i)
{
      Console.WriteLine("Копируем " + i);
      return i.Clone();
}

static int Main()
{
      CloneClass<int> ci = new CloneClass<int>();
      CloneClass<string> cs = new CloneClass<string>();

      // Копируем GenericTypesSample.Program+CloneClass`1[System.Int32]
      CreateCopy(ci);
      // Копируем GenericTypesSample.Program+CloneClass`1[System.String]
      CreateCopy(cs);
      return 0;
}
```

Универсальные типы инвариантны. Другими словами, если имеем класс Vector<производный класс>, при попытке представить его как Vector<базовый класс> возникнет ошибка компиляции:

```
static void OutVectorS(Vector<string> v)
{
      for (int i = 0; i < v.Length; i++)
      {
            Console.WriteLine(v[i]);
      }
}

static void OutVectorO(Vector<object> v)
{
      for (int i = 0; i < v.Length; i++)
      {
            Console.WriteLine(v[i]);
      }
}

static int Main()
{
      Vector<string> vs = new Vector<string>(3);

      OutVectorS(vs);
      OutVectorO(vs); // Ошибка
      return 0;
}
```

5.1.2. Ограничения параметров типа

При определении универсального члена можно ограничить виды типов, которые могут использоваться клиентским кодом в качестве аргументов типа при инициализации соответствующего типа или вызове метода. При попытке клиентского кода нарушить ограничения возникает ошибка компиляции. Это называется *ограничениями параметров типа*. Ограничения определяются с помощью контекстно-зависимого ключевого слова **where**:

```
      <ограничения параметров типа> :: <список ограничений1> [<список
ограничений2>] [...]

      <список ограничений> :: where <идентификатор> : <первичное ограничение>
[, <вторичные ограничения>] [, <ограничение конструктора>]
      <список ограничений> :: where <идентификатор> : <вторичные ограничения>
[, <ограничение конструктора>]
      <список ограничений> :: where <идентификатор> : <ограничение
конструктора>

      <первичное ограничение> :: struct
      <первичное ограничение> :: class
      <первичное ограничение> :: <тип класса>

      <вторичные ограничения> :: <вторичное ограничение> [, ...]
      <вторичное ограничение> :: <тип интерфейса>
      <вторичное ограничение> :: <идентификатор>

      <ограничение конструктора> :: new()
```

Таким образом, можно выделить шесть видов ограничений (табл. 5.1).

Табл. 5.1 – Виды ограничений параметров типа

Ограничение	Описание
where T : **struct**	Фактический тип должен быть типом по значению (кроме обнуляемых типов)
where T : **class**	Фактический тип должен быть ссылочным (классом, интерфейсом, делегатом, массивом и т.п.)
where T : **new**()	Фактический тип должен иметь открытый (**public**) конструктор по умолчанию (без параметров)
where T : <тип класса>	Фактический тип должен являться указанным классом или быть производным от него. Класс не должен быть Object, Array, Delegate, Enum или ValueType, изолированным или статическим
where T : <тип интерфейса>	Фактический тип должен являться указанным интерфейсом или реализовывать его

where T : U	Фактический параметра типа T должен являться типом фактического параметра U или быть производным от него (неприкрытое ограничение типа)

К одному параметру типа может применяться несколько ограничений, при этом сами ограничения могут быть универсального типа. При наложении ограничений на параметр типа увеличивается число допустимых операций и вызовов методов, поддерживаемых ограничивающим типом и всеми типами в его иерархии наследования. Пример:

```
class MyClass1<T> where T : class
{
      T a;
      public MyClass1(T b)
      {
            a = new T(); // Ошибка
            a = b;
      }
}

class MyClass2<T> where T : new()
{
      public T a = new T(); // ОК
}

class MyClass3<T> where T : Int32, Int64 { } // Ошибка
class MyClass4<T> where T : String { } // Ошибка

interface IRandomize<T>
{
      T Random();
}

class Rnd : IRandomize<int>
{
      static double Last = 0;

      public int Random()
      {
            double r;
            if (Last == 0) Last = DateTime.Now.Ticks % 1000000;
            Last = Math.Sqrt(Last);
            r = Last - (int)Last;
            Last = r*1000000.0;
            return (int)(r*int.MaxValue);
      }
}

class StdRandom : IRandomize<double>
{
      public static Random r = new Random();

      public double Random()
      {
            return r.NextDouble();
```

```
            }
      }

      class MyClass5<T, U>
            where T : IRandomize<U>, new()
            where U : IComparable
      {
            private T randomizer;

            public MyClass5()
            {
                  randomizer = new T();
            }

            public bool CompareToRandom(U min, U max, out U rnd)
            {
                  rnd = randomizer.Random();
                  return rnd.CompareTo(min) >= 0 && rnd.CompareTo(max) <= 0;
            }
      }

      class Program
      {
            static int Main()
            {
                  MyClass1<string> cls1 = new MyClass1<string>("!!!"); // OK
                  MyClass1<int?> cls2 = new MyClass1<int?>(null); // Ошибка

                  MyClass2<int> cls3 = new MyClass2<int>(); // OK
                  MyClass2<MyClass1<object>> cls4 = new
MyClass2<MyClass1<object>>(); // Ошибка

                  MyClass5<Rnd, int> cls5 = new MyClass5<Rnd, int>(); // OK
                  MyClass5<StdRandom, int> cls6 = new MyClass5<StdRandom,
int>(); // Ошибка
                  MyClass5<StdRandom, double> cls7 = new MyClass5<StdRandom,
double>(); // OK
                  int irnd;
                  double frnd;
                  bool bi, bf;

                  for (int i = 0; i < 10; i++)
                  {
                        bi = cls5.CompareToRandom(500000000, 1500000000, out
irnd);
                        bf = cls7.CompareToRandom(0.25, 0.75, out frnd);
                        Console.WriteLine(bi ? "{0} є [500000000, 1500000000]"
: "{0} є [0, 499999999]U[1500000001, 2147483647]", irnd);
                        Console.WriteLine(bf ? "{0} є [0.25, 0.75]" : "{0} є
[0, 0.25)U(0.75, 1]", frnd);
                  }

                  return 0;
            }
      }
```

Ошибки:

1) В типе T не гарантировано наличие конструктора по умолчанию;

2) Int32 и Int64 – структуры, а не классы;

3) Использование класса String в ограничениях запрещено;

4) Обнуляемый тип не является ссылочным;

5) В классе MyClass1<**object**> нет конструктора по умолчанию;

6) Класс StdRandom не наследует интерфейс IRandomize<**int**>.

В случае применения ограничения «**where** T : **class**» следует избегать использования операторов «==» и «!=» с параметром типа, потому что эти операторы выполняют только сравнение ссылок. Это верно даже в том случае, если эти операторы перегружаются в типе, который используется в качестве аргумента. Причиной такого поведения является то, что во время компиляции компилятор знает только, что T является ссылочным типом, и поэтому должен использовать операторы по умолчанию, которые действительны для всех ссылочных типов. Если необходимо проверять равенство содержимого объектов, рекомендуется применить ограничение «**where** T : IComparable<T>».

Если интерфейс указан в качестве ограничения параметра типа, могут использоваться лишь типы, реализующие интерфейс.

Универсальные типы, унаследованные от открытых базовых типов, должны указывать ограничения, которые соответствуют ограничениям базового типа или являются более строгими:

```
      class MyClass6<T> where T : IEnumerable, ICollection, new() { }
      class MyClass7<T> : MyClass6<T> where T : new() { } // Ошибка
      class MyClass8<T> : MyClass6<T> where T : IEnumerable<T>,
ICollection<T>, new() { } // Ошибка
      class MyClass9<T> : MyClass6<T> where T : IList, new() { } // OK
      class MyClass10<T> : MyClass6<T> where T : IList { } // Ошибка
```

Комментарии:

1) Класс MyClass7 не гарантирует, что тип T реализует интерфейсы IEnumerable и ICollection, а этого требует класс MyClass6.

2) Класс MyClass8 не гарантирует, что тип T реализует интерфейс ICollection. Интерфейс IEnumerable<T> наследуется от IEnumerable, поэтому это ограничение выполнено. Но интерфейс ICollection<T> наследует только IEnumerable<T> и IEnumerable, а ICollection – нет.

3) Ограничения класса MyClass9 описаны правильно, т.к. интерфейс IList наследует интерфейсы IEnumerable и ICollection.

4) Класс MyClass10 не гарантирует, что тип T будет иметь конструктор по умолчанию, а этого требует класс MyClass6.

 Пример: Samples\5.1\5_1_2_where.

5.1.3. Стандартные универсальные типы

Универсальные типы предоставляют решение проблем более ранних версий среды CLR и языка C#, в которых обобщения достигались за счет приведения типов к универсальному базовому типу Object и из него. Путем создания универсального класса можно создать коллекцию, которая была бы строго типизирована во время компиляции.

Ограничения, возникающие при использовании конкретных классов коллекции, можно продемонстрировать класса коллекции ArrayList. Это очень удобный класс коллекции, который можно использовать без изменений для хранения элементов как типов по значению, так и ссылочных типов. Однако, за удобство приходится платить. Любые типы, добавляемые к классу коллекции ArrayList, неявно приводятся к Object. Если в качестве элементов выступают типы по значению, то для них необходимо выполнять операции упаковки и распаковки (см. п. 3.1.5). Операции приведения, упаковки и распаковки снижают производительность. Еще одно ограничение состоит в отсутствии проверки типа во время компиляции. Поскольку класс коллекции ArrayList приводит все элементы к типу Object, невозможно предотвратить выполнение клиентским кодом во время компиляции действий, связанных с неверной интерпретацией типов элементов.

Поэтому необходимо для класса коллекции ArrayList и других подобных классов реализовать возможность указания для каждого отдельного экземпляра в клиентском коде определенного типа данных, который они должны использовать. Это устранит необходимость приведения к типу System.Object и даст компилятору возможность проверки типа. Другими словами, классу коллекции ArrayList необходим параметр типа. Таким вариантом этого класса является коллекция List<T>. С помощью данного класса можно создать список, который будет не только безопаснее, чем ArrayList, но и существенно быстрее, особенно если элементами списка являются экземпляры типов по значению.

Библиотека классов платформы .NET Framework содержит несколько новых универсальных классов коллекций в пространстве имен System.Collections.Generic. Их следует использовать по мере возможности вместо таких классов, как ArrayList в пространстве имен System.Collections.

§ 5.2. Потоки

Для разделения различных выполняемых приложений в операционных системах используются *процессы*. *Потоки* являются основными элементами, для которых операционная система выделяет время процессора; внутри процесса может выполняться более одного потока. Каждый поток поддерживает обработчик исключений, планируемый *приоритет* и набор структур, используемых системой для сохранения *контекста потока* во время его планирования. Контекст потока содержит все необходимые данные для возобновления выполнения (включая набор регистров процессора и стек) в адресном пространстве ведущего процесса.

5.2.1. Основы организации потоков

По умолчанию, все разрабатываемые нами приложения являются *однопоточными*. *Многопоточность* позволяет приложениям разделять задачи и работать над каждой независимо, чтобы максимально эффективно задействовать процессор и пользовательское время. Однако, чрезмерное злоупотребление многопоточностью может снизить эффективность программы. Разделять процесс на потоки следует только в том случае, если это оправданно.

5.2.1.1. Потоки и многозадачность

Поток является единицей обработки данных, а *многозадачность* – это одновременное исполнение нескольких потоков. Существует два вида многозадачности – совместная (cooperative) и вытесняющая (preemptive). Самые ранние версии Microsoft Windows поддерживали совместную многозадачность. Это означало, что каждый поток отвечал за возврат управления процессору, чтобы тот смог обработать другие потоки. Т.е., если какой-либо поток не возвращал управление (из-за ошибки в программе или по другой причине), другие потоки не могли продолжить выполнение. И если этот поток «зависал», то «зависала» вся система.

Однако, начиная с Windows NT, стала поддерживаться вытесняющая многозадачность. При этом процессор отвечает за выдачу каждому потоку определенного количества времени, в течение которого поток может выполняться – *кванта времени* (timeslice). Далее процессор переключается между

разными потоками, выдавая каждому потоку его квант времени, а программист может не заботится о том, как и когда возвращать управление, в результате чего могут работать и другие потоки.

Даже в случае вытесняющей многозадачности, если в системе установлен только один процессор (вернее будет сказать – одно процессорное ядро, т.к. современные процессоры содержат несколько ядер), то все равно в любой момент времени реально будет исполняться только один поток. Поскольку интервалы между переключениями процессора от процесса к процессу измеряются миллисекундами, возникает иллюзия многозадачности. Чтобы несколько потоков на самом деле работали одновременно, нам потребуется работать на многопроцессорной или многоядерной машине (или использовать процессоры, поддерживающие технологию Intel HyperThreading, позволяющую двум потокам работать на одном процессорном ядре параллельно), а также использовать специальные компиляторы, позволяющие назначить каждому потоку выполнение на соответствующем ядре процессора.

5.2.1.2. Переключение контекста

Неотъемлемый атрибут потоков – *переключение контекста* (context switching). Процессор с помощью аппаратного таймера определяет момент окончания кванта, выделенного для данного потока. Когда аппаратный таймер генерирует прерывание, процессор сохраняет в стеке содержимое всех регистров для данного потока. Затем процессор перемещает содержимое этих же регистров в специальную структуру данных контекста. При необходимости переключения обратно на поток, выполнявшийся прежде, процессор выполняет обратную процедуру и восстанавливает содержимое регистров из структуры контекста, ассоциированной с потоком. Весь этот процесс называется переключением контекста.

5.2.1.3. Правила использования потоков

После создания многопоточной программы, мы можем неожиданно обнаружить, что издержки, связанные с созданием и диспетчеризацией потоков, могут привести к тому, что однопоточное приложение работает быстрее.

Например, если нам надо считать с диска три файла, создание для этого трех потоков не принесет пользы, поскольку все они будут обращаться к одному и тому же жесткому диску. Поэтому всегда нужно стараться тестиро-

вать обе версии программы и выбирать оптимальное решение.

Потоки следует использовать тогда, когда мы стремимся достичь повышенного параллелизма, упростить структуру программы и эффективнее использовать процессорное время.

1) Повышенный параллелизм. Если мы написали сложный алгоритм обработки каких-либо данных (сортировка больших массивов данных, обработка больших блоков текста, сканирование файловой системы и т.п.), время выполнения которого может превышать 5-10 секунд, то логично будет выделить работу этого алгоритма в отдельный поток. Иначе во время работы программы диспетчер сообщений, поступающих от ОС, не будет получать управления и, таким образом, пользовательский ввод в программе окажется полностью блокированным. Хотя мы знаем, что некоторые файловые менеджеры (Total Commander и т.п.), архиваторы (WinRAR и т.п.), менеджеры закачек, браузеры, программы для обслуживания жестких дисков, программы редактирования мультимедиа и т.д. допускают выполнение трудоемких задач в фоновом режиме. В этом случае, при необходимости, такую задачу можно прервать, или запустить параллельно другую задачу.

Кроме того, сама выполнение самой задачи, если оно подразумевает обработку однородных данных, может быть разделено между несколькими потоками.

2) Упрощенная структура. Например, наше приложение получает асинхронные сообщения (т.е. которые могут приходить в любой момент, в т.ч. во время обработки других сообщений) от других приложений или потоков. Популярный способ упрощения структуры таких систем – использование очередей и асинхронной обработки. Вместо прямого вызова методов создаются специальные объекты сообщений и помещаются в очереди, в которых производится их обработка. На другом конце этих очередей работает несколько потоков, настроенных на отслеживание приходящих сообщений.

3) Эффективное использование процессорного времени. Часто приложение реально не выполняет никакой работы, в то же время продолжая использовать свой квант процессорного времени. Например, ожидает поступления какого-либо события (от устройства ввода команд пользователя, об окончании печати документа, об окончании обработки файла и т.д.). Эти случаи являются кандидатами на перевод в потоки, работающие в фоновом режиме.

5.2.2. Работа с потоками

Создание потоков и управление ими осуществляется с помощью класса System.Threading.Thread. С него мы и начнем.

5.2.2.1. Многопоточное приложение .NET

Прежде чем изучать способы использования потоков в .NET, посмотрим, как создать простейший вторичный поток на языке C#:

```
using System;
using System.Threading;

namespace SimpleThreadSample
{
      class Program
      {
            static void ThreadMethod()
            {
                  Console.WriteLine("Поток работает");
            }

            static int Main()
            {
                  ThreadStart ts = new ThreadStart(ThreadMethod);
                  Thread th = new Thread(ts);

                  Console.WriteLine("Main: запускаем поток");
                  th.Start();
                  Console.WriteLine("Main: поток запущен");
                  return 0;
            }
      }
}
```

Вывод на консоль:

```
Main: запускаем поток
Main: поток запущен
Поток работает
```

Мы видим, что сообщение метода Main выводится перед сообщением потока. Это доказывает, что поток действительно работает асинхронно. Проанализируем происходящие здесь события.

В пространстве имен System.Threading описаны типы, необходимые для организации потоков в среде .NET. Тип ThreadStart – это делегат, который вызывается при запуске потока. Его необходимо указать в конструкторе класса Thread, собственно инкапсулирующего поток. Описание этого делегата следующее:

```
delegate void ThreadStart();
```

Как видим, требуемый метод должен ничего не возвращать и не иметь параметров. Есть и другой делегат ParameterizedThreadStart для методов с параметром:

```
delegate void ParameterizedThreadStart(object obj);
```

Его также можно использовать при конструировании экземпляра класса Thread. Метод Start объекта Thread запускает поток на выполнение.

 Пример: Samples\5.2\5_2_2_simple.

5.2.2.2. Домены приложений

В .NET потоки работают в областях под названием *домены приложений*. Домены аналогичны процессам Win32 в том смысле, что предлагают те же возможности. Но в Win32 поток привязан к единственному процессу. Поток одного процесса не может вызывать метод потока, принадлежащий другому процессу. Однако в .NET потоки могут пересекать границы доменов, а метод из одного потока может вызывать метод из другого домена. Поэтому более удачное определение домена приложений звучит так: это логический процесс внутри физического процесса.

В .NET функциональность доменов приложений инкапсулирована в классе AppDomain. Программист может создавать собственные домены и настраивать их, загружать в них сборки и выгружать, когда они больше не нужны. Небольшой пример:

```
      AppDomain domain = AppDomain.CreateDomain("MyDomain");

      Console.WriteLine("Создаем новый домен приложений");
      Console.WriteLine("Главный домен: " +
AppDomain.CurrentDomain.FriendlyName);
      Console.WriteLine("Дочерний домен: " + domain.FriendlyName);
```

Вывод на консоль:

```
      Создаем новый домен приложений
      Главный домен: 5_2_2_domain.exe
      Дочерний домен: MyDomain
```

Подробную информацию о классе AppDomain можно получить в библиотеке MSDN.

 Пример: Samples\5.2\5_2_2_domain.

5.2.2.3. Класс Thread

Практически все операции с потоками инкапсулированы в классе Thread:

1) Создание потоков и объектов Thread. Мы можем получить новый поток, создав экземпляр класса Thread как в примере выше. Другой способ получить объект Thread для потока, исполняемого в данный момент – вызов статического метода Thread.CurrentThread:

```
Thread th = Thread.CurrentThread;

Console.WriteLine("Текущий поток: ");
Console.WriteLine("  Язык: {0}", th.CurrentCulture);
Console.WriteLine("  Имя: {0}", th.Name);
Console.WriteLine("  Приоритет: {0}", th.Priority);
Console.WriteLine("  Состояние: {0}", th.ThreadState);
Console.WriteLine();
```

2) Уничтожение потоков. Уничтожить поток можно вызовом метода Thread.Abort. Исполняющая среда принудительно завершает выполнение потока, генерируя исключение ThreadAbortException. Даже если исполняемый метод потока попытается уловить ThreadAbortException, исполняющая среда этого не допустит. Однако она исполнит код из блока finally потока, выполнение которого прервано, если этот блок присутствует.

Необходимо отдавать себе отчет в том, что при вызове метода Thread.Abort выполнение потока не может остановиться сразу. Исполняющая среда ожидает, пока поток не достигнет *безопасной точки* (safe point). Поэтому, если наша программа зависит от некоторых действий, которые происходят после прерывания потока, и надо быть уверенными в том, что поток остановлен, необходимо использовать метод Thread.Join. Это синхронный вызов, т.е. он не вернет управление, пока поток не будет остановлен.

После прерывания поток нельзя перезапустить. В этом случае, несмотря на то, что у нас есть экземпляр класса Thread, от него нет никакой пользы в плане выполнения кода.

3) Управление временем существования потоков. Для того, чтобы приостановить («усыпить») текущий поток, используется метод Thread.Sleep, который принимает единственный аргумент, представляющий собой время (в миллисекундах) «сна» потока.

Есть еще два способа вызова метода Thread.Sleep. Первый – вызов Thread.Sleep со значением 0. При этом мы заставляем текущий поток освобо-

дить неиспользованный остаток своего кванта процессорного времени. При передаче значения Timeout.Infinite поток будет «усыплен» на неопределенно долгий срок, пока это состояние потока не будет прервано другим потоком, вызвавшим метод приостановленного потока Thread.Interrupt. В этом случае поток получит исключение ThreadInterruptedException.

Пример:

```
      static void ThreadMethod1(object id)
      {
            try
            {
                  Console.WriteLine(id);
                  Thread.Sleep(Timeout.Infinite);
            }
            catch (ThreadInterruptedException)
            {
            }
      }

      static int Main()
      {
            Thread th1 = new Thread(new
ParameterizedThreadStart(ThreadMethod1));

            Console.WriteLine("Запуск вторичного потока th1");
            th1.Start(1);
            Thread.Sleep(5000);
            Console.WriteLine("Состояние th1: {0}", th1.ThreadState);
            Console.WriteLine("Прерываем вторичный поток th1");
            th1.Interrupt();
            Console.WriteLine("Состояние th1: {0}", th1.ThreadState);
            th1.Join();
            Console.WriteLine("Состояние th1: {0}", th1.ThreadState);
            return 0;
      }
```

Вывод на консоль:

```
      Состояние th1: WaitSleepJoin
      Прерываем вторичный поток th1
      Состояние th1: WaitSleepJoin
      Состояние th1: Stopped
```

В этом примере поток выводит на экран переданное ему значение (1) и «засыпает» на неограниченное время. Его состояние – WaitSleepJoin. В этом состоянии находится поток, вызвавший метод Sleep или Join. При прерывании потока извне методом Interrupt генерируется исключение ThreadInterruptedException, т.о. поток может отреагировать на прерывание. Если его прервать методом Abort, данное исключение не будет генерироваться. Затем выводим на экран состояние потока. Это все еще WaitSleepJoin – поток не достиг безопасной точки и пока не прерван (хотя, на других компь-

ютерах может оказаться, что поток уже достиг безопасной точки и успел прерваться). Поэтому вызываем метод Join. После того, как он вернет управление первичному потоку, видим, что вторичный поток действительно остановлен.

Второй способ приостановить исполнение потока – вызов метода Thread.Suspend. Между этими методами есть несколько важных отличий. Во-первых, можно вызвать метод Thread.Suspend для любого потока, а не только текущего. Во-вторых, если таким образом приостановить выполнение потока, любой другой поток способен возобновить его выполнение с помощью метода Thread.Resume. Единственный вызов Thread.Resume возобновит исполнение данного потока независимо от числа вызовов метода Thread.Suspend, выполненных ранее:

```
      static void ThreadMethod2(object id)
      {
            while (true)
            {
                  Console.Write(id);
                  Thread.Sleep(500);
            }
      }

      static int Main()
      {
            Thread th2 = new Thread(new
ParameterizedThreadStart(ThreadMethod2));

            Console.WriteLine("Запуск вторичного потока th2");
            th2.Start(2);
            Thread.Sleep(5000);
            Console.WriteLine("\nСостояние th2: {0}", th2.ThreadState);
            Console.WriteLine("Приостанавливаем вторичный поток th2");
            th2.Suspend();
            Console.WriteLine("Состояние th2: {0}", th2.ThreadState);
            Thread.Sleep(2000);
            Console.WriteLine("Возобновляем вторичный поток th2");
            th2.Resume();
            Thread.Sleep(5000);
            Console.WriteLine("\nУничтожаем вторичный поток th2");
            th2.Abort();
            th2.Join();
            return 0;
      }
```

Вывод на консоль:

```
Запуск вторичного потока th2
22222222222
Состояние th2: Running
Приостанавливаем вторичный поток th2
Состояние th2: SuspendRequested, WaitSleepJoin
Возобновляем вторичный поток th2
```

```
22222222222
Уничтожаем вторичный поток th2
```

Результаты работы на другой машине также могут отличаться. Например, при первом выводе состояния потока можем получить не Running, а WaitSleepJoin, если в это время поток будет «спать». Также до вызова метода «th2.Suspend()» поток может еще раз успеть вывести на консоль «2» в какой-либо строке. Это лишний раз доказывает, что синхронизация и планирование потоков – достаточно сложные задачи. О них мы поговорим далее.

Более того, компилятор .NET версии 3.5 уже считает методы Suspend и Resume устаревшими (в программе отключено предупреждение CS0618, иначе мы бы увидели это). Для управления потоками рекомендуется использовать такие объекты, как мониторы, семафоры и т.п. (классы Monitor, Mutex, Event, Semaphore).

 Пример: Samples\5.2\5_2_2_thread.

5.2.2.4. Планирование потоков

Переключение процессора на следующий поток выполняется не произвольным образом. У каждого потока есть приоритет, указывающий процессору, как должно планироваться выполнение этого потока по отношению к другим потокам системы. Для потоков, создаваемых в период выполнения, уровень приоритета по умолчанию равен Normal. Для просмотра и установки этого значения служит свойство Thread.Priority. Установщик свойства Thread.Priority принимает аргумент типа Thread.ThreadPriority, представляющего собой перечисление. Допустимые значения – Highest, AboveNormal, Normal, BelowNormal и Lowest (в порядке убывания приоритета).

Пример:

```
const int Counter = 10000;

static void ThreadMethod(object id)
{
      while (true)
      {
            for (int j = 0; j <= Counter; j++)
            {
                  for (int k = 0; k <= Counter; k++)
                  {
                        if (j == Counter && k == Counter)
                        {
                              Console.Write(id);
                        }
                  }
```

```
                }
            }
        }

        static int Main()
        {
            Thread[] th;
            ThreadPriority[] priority = {
                ThreadPriority.Highest,
                ThreadPriority.Lowest,
            };

            th = new Thread[priority.Length];

            for (int i = 0; i < th.Length; i++)
            {
                th[i] = new Thread(ThreadMethod);
                th[i].Priority = priority[i];
                th[i].Start(i + 1);
            }

            return 0;
        }
```

В данном примере количество запускаемых потоков можно регулировать, добавляя или удаляя элементы из массива priority, а количество итераций циклов – изменяя значение константы Counter. Зачем это сделано? Результаты работы программы зависят от частоты процессоров и количества процессорных ядер, установленных в системе. Если вывод данных происходит слишком медленно, необходимо уменьшить значение счетчика Counter, если слишком быстро – увеличить его. Бесполезное, в принципе, условие «**if** (j == Counter && k == Counter)» (можно ведь было вызов метода WriteLine просто разместить после циклов **for**) требуется для того, чтобы процессор или компилятор не оптимизировали пустые циклы.

Например, создадим два потока с приоритетами Highest и Lowest. Смотрим на результаты работы (прервать выполнение программы можно нажатием комбинации клавиш Ctrl+C):

```
121212121212121212121212121212121212121212121212121212121212121212...
```

Несмотря на разницу в приоритетах, потоки получают одинаковое количество процессорного времени! Дело в том, что программа была запущена на ПК, имеющем один процессор с двумя ядрами, технология HyperThreading была отключена. В итоге каждый процесс мог получить в свое распоряжение по одному ядру, и конкуренции между ними не было (рис. 5.1).

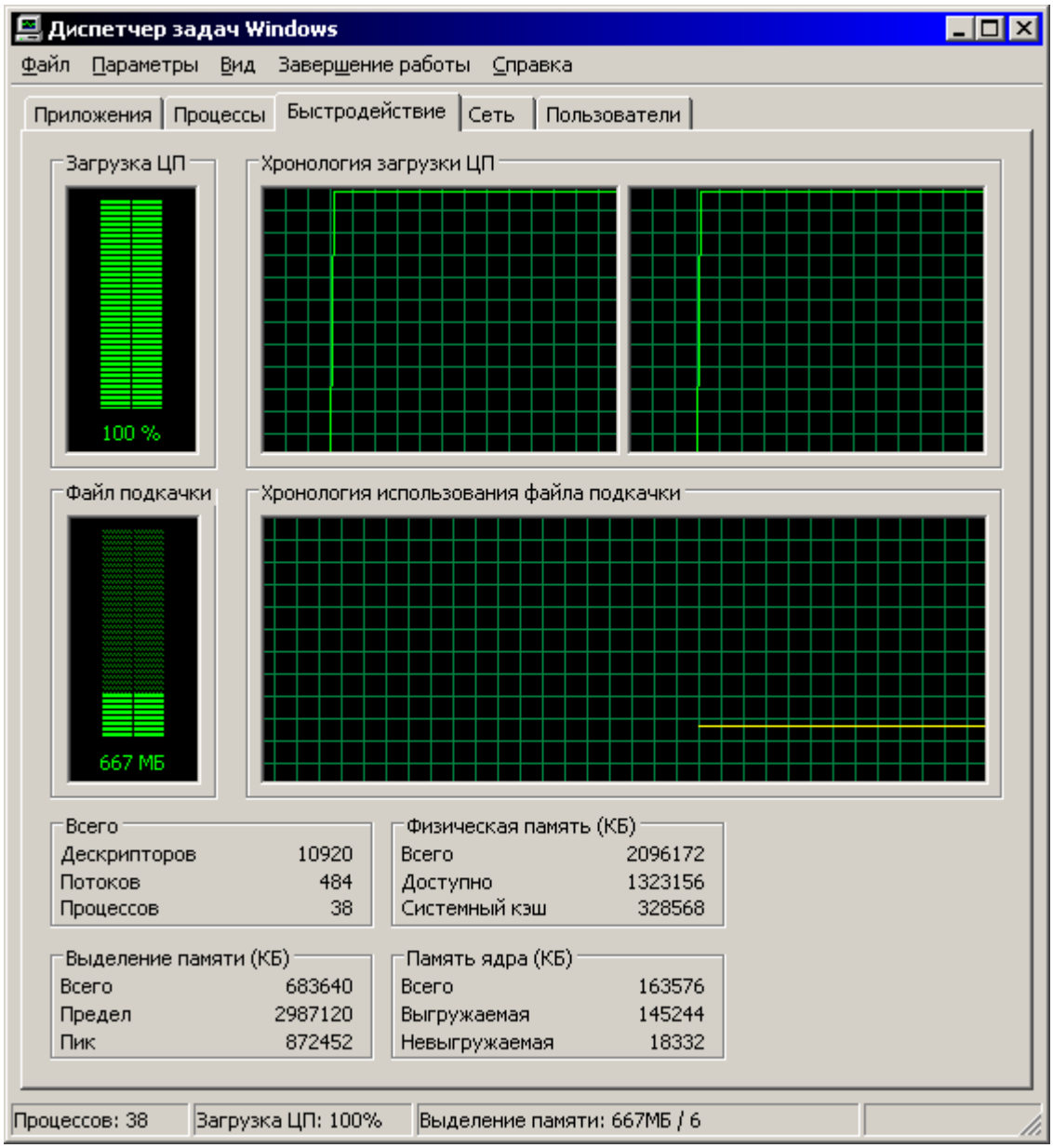


Рис. 5.1 – Диспетчер задач Windows

Каждый поток полностью нагрузил свое ядро, в результате чего загрузка процессора достигла 100%.

По умолчанию, операционная система позволяет приложению использовать все имеющиеся процессорные ядра. Однако можно вручную задать соответствие между запущенными процессами и доступными для них ядрами. Для этого необходимо вызвать контекстное меню для интересующего процесса в диспетчере задач, выбрать в нем пункт «Задать соответствие…» и в появившемся диалоге (рис. 5.2) пометить только разрешенные для процесса ядра.

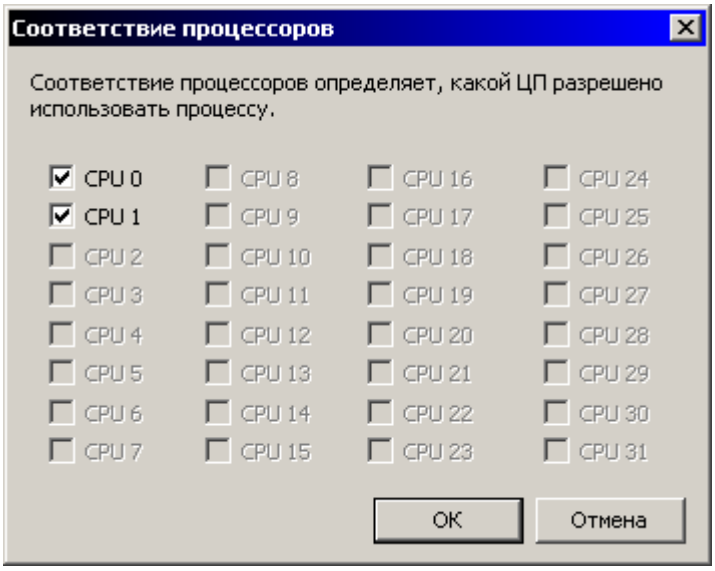


Рис. 5.2 – Соответствие процесса и процессоров

Изменим условия задачи, добавив еще два потока:

```
ThreadPriority[] priority = {
      ThreadPriority.Highest,
      ThreadPriority.Lowest,
      ThreadPriority.Normal,
      ThreadPriority.Normal,
};
```

Результаты работы программы:

```
131134114311431141314131411341411341134131411231413141314141314131413...
```

В данном случае второй поток процессорного времени практически не получает, второй и третий получают его примерно поровну (имея одинаковый приоритет). Поэтому регулируйте в этом примере количество потоков и их приоритет, исходя из имеющейся аппаратной конфигурации.

Изменим условие задачи еще раз. Вместо бесконечного цикла **while** сделаем цикл

```
for (int i = 0; i < 15; i++)
```

Результаты работы программы:

```
131411431143131141314131414134343434343434343222222222222222
```

Первым свою работу завершил, как и ожидалось, первый поток, а последним – второй. Третий и четвертый потоки расположены между ними – в зависимости от ситуации, первым завершит свою работу либо один из них, либо второй.

Помните, что когда мы указываете процессору приоритет для потока,

то его значение используется ОС как часть алгоритма планирования распределения процессорного времени. В .NET этот алгоритм основан на уровнях приоритета Thread.Priority, а также на *классе приоритета* (priority class) процесса и значении *динамического повышения приоритета* (dynamic boost). С помощью всех этих значений создается численное значение (для процессоров архитектуры x86 – от 0 до 31), представляющее реальный приоритет потока.

Класс приоритета процесса задается свойством PriorityClass, а динамическое повышение приоритета – свойством PriorityBoostEnabled класса System.Diagnostics.Process. Например:

```
using System;
using System.Diagnostics;

class Program
{
      static int Main()
      {
            Process proc = Process.GetCurrentProcess();

            proc.PriorityClass = ProcessPriorityClass.AboveNormal;
            proc.PriorityBoostEnabled = true;
            Console.WriteLine(proc.BasePriority);
            return 0;
      }
}
```

Класс приоритета процесса определяет *базовый приоритет процесса* (в табл. 5.2 перечислены в порядке возрастания приоритета). Базовый приоритет процесса можно узнать, используя доступное только для чтения свойство BasePriority класса Process.

Табл. 5.2 – Класс приоритета и базовый приоритет процесса

Класс приоритета	Базовый приоритет
Idle	4
BelowNormal	6
Normal	8
AboveNormal	10
High	13
RealTime	24

Реальный приоритет потока получается сложением базового приоритета процесса и собственного приоритета потока.

Свойство PriorityBoostEnabled используется для временного увеличения уровня приоритета потоков, взятых из состояния ожидания. Приоритет

сбрасывается при возвращении процесса в состояние ожидания.

 Пример: Samples\5.2\5_2_2_priority.

Несколько потоков с одинаковым приоритетом получают равное количество процессорного времени. Это называется *циклическим планированием* (round robin scheduling).

5.2.2.5. Пул потоков

Если имеются небольшие задачи, которые нуждаются в фоновой обработке, *пул управляемых потоков* – это самый простой способ воспользоваться преимуществами нескольких потоков. Статический класс ThreadPool обеспечивает приложение пулом рабочих потоков, управляемых системой, позволяя пользователю сосредоточиться на выполнении задач приложения, а не на управлении потоками.

Потоки из пула потоков являются фоновыми потоками. Для каждого потока используется размер стека и приоритет по умолчанию. Для каждого процесса можно использовать только один пул потоков.

Количество операций, которое может быть помещено в очередь пула потоков, ограничено только объемом памяти, однако пул потоков ограничивает количество потоков, которое может быть одновременно активно в процессе. По умолчанию это ограничение составляет 500 рабочих потоков на ЦП и 1000 потоков асинхронного ввода/вывода (зависит от версии .NET Framework). Можно управлять максимальным количеством потоков с помощью методов GetMaxThreads и SetMaxThreads. Также можно задавать минимальное количество потоков с помощью методов GetMinThreads и SetMinThreads. Пользоваться этими методами следует с осторожностью, т.к. если задать большое минимальное число потоков, и часть из них не будет использована, это приведет к ненужному расходу системных ресурсов.

Добавление в пул рабочих потоков производится путем вызова метода QueueUserWorkItem и передачи делегата WaitCallback, представляющего метод, который выполняет задачу:

```
static bool QueueUserWorkItem(WaitCallback callBack);
static bool QueueUserWorkItem(WaitCallback callBack, object state);
delegate void WaitCallback(object state);
```

Пример:

```
const int Counter = 10000;
```

```
static void ThreadMethod(object id)
{
      for (int i = 0; i < 15; i++)
      {
            for (int j = 0; j <= Counter; j++)
            {
                  for (int k = 0; k <= Counter; k++)
                  {
                        if (j == Counter && k == Counter)
                        {
                              Console.Write(id);
                        }
                  }
            }
      }
}

static int Main()
{
      int min_cpu, min_io, max_cpu, max_io;

      ThreadPool.GetMinThreads(out min_cpu, out min_io);
      ThreadPool.GetMaxThreads(out max_cpu, out max_io);
      Console.WriteLine("Потоки ЦП: {0}..{1}", min_cpu, max_cpu);
      Console.WriteLine("Асинхронные потоки: {0}..{1}", min_io, max_io);

      for (int i = 0; i < 4; i++)
      {
            ThreadPool.QueueUserWorkItem(ThreadMethod, i + 1);
      }

      Console.ReadKey(true);
      return 0;
}
```

Вывод на консоль:

```
Потоки ЦП: 2..500
Потоки ввода-вывода: 2..1000
12121212112313121312132413241342341234124342434243434343
```

Результаты аналогичны предыдущему примеру, только приоритеты всех потоков одинаковы (Normal). Здесь в пуле запускаются четыре потока, но из-за ограничений ЦП (2 ядра) первыми выполняются 1-й и 2-й потоки, а уже затем 3-й и 4-й. Окончание работы главного потока приведет к прерыванию работы всех потоков в пуле. Поэтому нажатие на любую клавишу прервет работу. В принципе, вместо вызова метода ReadKey можно «усыпить» основной поток, например, вызвать метод Thread.Sleep(5000). Тогда на выполнение потоков в пуле будет отведено 5 секунд. Если же мы хотим гарантированно дождаться выполнения работы всеми потоками, используем метод, возвращающий количество доступных потоков (GetAvailableThreads):

```
int cur_cpu, cur_io;
```

```
do
{
      Thread.Sleep(100);
      ThreadPool.GetAvailableThreads(out cur_cpu, out cur_io);
} while (cur_cpu != max_cpu);

Console.WriteLine("\nВсе потоки завершили свою работу");
```

Заметим, что в цикле основной поток «усыпляется» на 100 мс. Это сделано для того, чтобы дать поработать другим потокам. Иначе постоянный запрос количества доступных потоков в цикле будет приводить к трате процессорного времени.

 Пример: Samples\5.2\5_2_2_queue.

Добавление в пул асинхронных потоков производится путем вызова метода RegisterWaitForSingleObject и передачи ему объекта синхронизации WaitHandle. Этот объект ждет наступления некоторого события, и при его наступлении или при истечении времени ожидания вызывает метод, представленный делегатом WaitOrTimerCallback:

```
static RegisteredWaitHandle RegisterWaitForSingleObject(
      WaitHandle waitObject, WaitOrTimerCallback callBack,
      object state, int timeOutInterval, bool executeOnlyOnce);
delegate void WaitOrTimerCallback(object state, bool timedOut);
```

Объект синхронизации – это экземпляр класса WaitHandle или его потомка:

```
System.Threading.WaitHandle
├── System.Threading.EventWaitHandle
│   ├── System.Threading.AutoResetEvent
│   └── System.Threading.ManualResetEvent
├── System.Threading.Mutex
└── System.Threading.Semaphore
```

Пример:

```
const int Counter = 10000;

class WaitObject
{
      public bool TimeOut = false;
      public int ID = 1;
      public RegisteredWaitHandle Handle;

      public static void CallbackMethod(object state, bool timeout)
      {
            WaitObject obj = (WaitObject)state;
            int id = obj.ID++;

            if (!timeout)
            {
```

```
                        Console.WriteLine("\nПоток #{0} получил сигнал о
запуске", id);
                        for (int i = 0; i < 15; i++)
                        {
                              for (int j = 0; j <= Counter; j++)
                              {
                                    for (int k = 0; k <= Counter; k++)
                                    {
                                          if (j == Counter && k == Counter)
                                          {
                                                Console.Write(id);
                                          }
                                    }
                              }
                        }
                  }
                  else
                  {
                        Console.WriteLine("\nВремя ожидания закончилось");
                        obj.TimeOut = true;
                        obj.Handle.Unregister(null);
                  }
            }
      }

      static int Main()
      {
            int max_cpu, max_io, cur_cpu, cur_io;
            WaitObject obj = new WaitObject();
            AutoResetEvent are = new AutoResetEvent(false);
            const int wait = 10;
            char key;

            Console.WriteLine("Для запуска потока в пуле нажмите S");
            Console.WriteLine("Для отмены ожидания новых потоков - U");
            Console.Write("Через {0} сек ожидание будет окончено", wait);
            obj.Handle = ThreadPool.RegisterWaitForSingleObject(are,
                  WaitObject.CallbackMethod, obj, wait * 1000, false);

            do
            {
                  if (Console.KeyAvailable)
                  {
                        key = Console.ReadKey(true).KeyChar;
                        if (key == 'S' || key == 's')
                        {
                              are.Set();
                        }
                        else if (key == 'U' || key == 'u')
                        {
                              obj.Handle.Unregister(null);
                              Console.WriteLine("\nОжидание отменено");
                              break;
                        }
                  }
                  else
                  {
                        Thread.Sleep(100);
                  }
            } while (!obj.TimeOut);
```

```
            ThreadPool.GetMaxThreads(out max_cpu, out max_io);
            ThreadPool.GetAvailableThreads(out cur_cpu, out cur_io);
            if (cur_io != max_io)
            {
                  Console.WriteLine("\nПодождите, пока все потоки завершат
свою работу");
                  do
                  {
                        Thread.Sleep(100);
                        ThreadPool.GetAvailableThreads(out cur_cpu, out
cur_io);
                  } while (cur_io != max_io);
                  Console.WriteLine("\nВсе потоки завершили свою работу");
            }

            Console.ReadKey(true);
            return 0;
      }
```

В данном примере потоки в пуле запускаются при поступлении сигнала. Сигнал эмулируется вызовом метода AutoResetEvent.Set() при нажатии на клавишу «S». При нажатии клавиши «U», или если в течение 10 секунд (значение константы wait) не поступают сигналы, регистрация новых сигналов прекращается (вызовом метода RegisteredWaitHandle.Unregister). Иначе основной поток «засыпает» на 100 мс чтобы, как уже было сказано, не потреблять системные ресурсы и дать поработать асинхронным потокам в пуле. Затем опрос клавиатуры повторяется.

Если время ожидания новых потоков истекло, то регистрация в методе CallbackMethod также отменяется, хотя это делать не обязательно. Если регистрацию при значении timeout = **false** не отменять, то новые сообщения будут продолжать приниматься. Спустя 10 секунд, если новых сообщений о регистрации не будет, CallbackMethod с параметром timeout = **false** будет вызван повторно, и т.д.

Для завершения всех потоков в пуле снова используем значения, возвращаемые методом GetAvailableThreads. Но проверяем не количество рабочих потоков, как в предыдущем примере, а количество асинхронных потоков ввода-вывода.

Пример вывода на консоль (зависит от нажатых клавиш):

```
      Для запуска потока в пуле нажмите S
      Для отмены ожидания новых потоков - U
      Через 10 сек ожидание будет окончено
      Поток #1 получил сигнал о запуске
      11
      Поток #2 получил сигнал о запуске
      Поток #3 получил сигнал о запуске
```

```
1233132312331
Поток #4 получил сигнал о запуске
2343321
Поток #5 получил сигнал о запуске
34521354321531243523514551254551524 55
Время ожидания закончилось
Подождите, пока все потоки завершат свою работу
2514521424244444
Все потоки завершили свою работу
```

Подробную информацию о данных классах и их членах смотрите в библиотеке MSDN.

 Пример: Samples\5.2\5_2_2_async.

5.2.3. Безопасность и синхронизация потоков

Что произойдет при попытке одновременного доступа к объекту нескольких потоков? Проверим:

```
using System;
using System.Text;
using System.Threading;

namespace ThreadSynchronizeSample
{
      class Program
      {
            static StringBuilder sb;

            static void ThreadMethod(object id)
            {
                  for (int i = 0; i < sb.Length; i++)
                  {
                        sb[i] = (char)id;
                        Thread.Sleep(100);
                  }
            }

            static int Main()
            {
                  Thread th1 = new Thread(ThreadMethod);
                  Thread th2 = new Thread(ThreadMethod);

                  sb = new StringBuilder("----------------------");
                  th1.Start(1);
                  th2.Start(2);
                  th1.Join();
                  th2.Join();
                  Console.WriteLine(sb);
                  return 0;
            }
      }
}
```

Вывод на консоль:

```
21111121121212121121
```

Два потока модифицировали содержимое одного и того же объекта, в результате чего его состояние стало неопределенным.

 Пример: Samples\5.2\5_2_3_sync.

Как избежать подобных непредсказуемых состояний? Существует стандартный способ решения этой проблемы – *синхронизация*. Синхронизация позволяет создавать *критические секции* (critical sections) кода, в которые в каждый отдельный момент может входить только один поток, гарантируя, что любые временные недействительные состояния вашего объекта будут невидимы его клиентам. Т.е. выполнение действий внутри критической секции является *атомарной операцией* с точки зрения других потоков.

Например, для асинхронных потоков в пуле мы использовали временную переменную «id = obj.ID++» в теле метода CallbackMethod. Если бы вместо id использовалось поле obj.ID, то не было бы гарантии, что во время работы этого метода для какого-либо потока значение поля не изменилось бы другим потоком. Но даже операция инкремента не является атомарной, т.к. состоит из трех шагов:

1. Загрузить значение из экземпляра переменной в регистр;
2. Увеличить значение регистра на 1;
3. Сохранить значение в экземпляре переменной.

Другой поток может вклиниться между любой из этих операций.

5.2.3.1. Защита кода с помощью класса Monitor

Статический класс System.Threading.Monitor контролирует доступ к объектам, предоставляя блокировку объекта одному потоку. Блокировки объектов предоставляют возможность ограничения доступа к критической секции. Пока поток владеет блокировкой для объекта, никакой другой поток не может ею завладеть. Для управления блокировкой чаще всего используются два метода:

```
static void Enter(object obj);
static void Exit(object obj);
```

Первый метод пытается получить блокировку монитора для указанного объекта. Если у другого потока уже есть эта блокировка, текущий поток блокируется до тех пор, пока блокировка не будет освобождена. Объект должен являться экземпляром ссылочного типа. Второй метод снимает блокировку.

Изменим метод ThreadMethod в примере выше:

```
Monitor.Enter(sb);
for (int i = 0; i < sb.Length; i++)
{
      sb[i] = (char)id;
      Thread.Sleep(100);
}
Console.WriteLine(sb);
Monitor.Exit(sb);
```

Теперь два потока не смогут одновременно изменять строку:

```
11111111111111111111
22222222222222222222
22222222222222222222
```

Поток с идентификатором «2» был запущен вторым, поэтому в итоге в строке остались двойки. В другой ситуации вторым может запуститься поток с идентификатором «1».

 Пример: Samples\5.2\5_2_3_monitor.

Для того, чтобы синхронизировать коллекции, следует в методы Enter и Exit передавать ссылку не на саму коллекцию, а на ее свойство SyncRoot, наследуемое от интерфейса ICollection. Например, класс Array реализует интерфейс ICollection, поэтому все массивы синхронизируются следующим образом:

```
int[] mas = new int[100];
Monitor.Enter(mas.SyncRoot);
// безопасная обработка массива
Monitor.Exit(mas.SyncRoot);
```

5.2.3.2. Применение блокировок монитора оператором lock

Блокировать объект также позволяет оператор языка C# **lock**. Вспомним его синтаксис (см. п. 3.4.1.1):

```
<оператор блокировки> :: lock (<выражение>) <внедряемый оператор>
```

Фактически, это синтаксическое сокращение для вызовов методов Monitor.Enter и Monitor.Exit в рамках блока try-finally:

```
Monitor.Enter(<выражение>);
try
{
      <внедряемый оператор>
}
finally
{
      Monitor.Exit(<выражение>);
}
```

Следующий пример даст результат, аналогичный предыдущему:

```
static StringBuilder sb;

static void ThreadMethod(object id)
{
      lock (sb)
      {
            for (int i = 0; i < sb.Length; i++)
            {
                  sb[i] = (char)id;
                  Thread.Sleep(100);
            }
            Console.WriteLine(sb);
      }
}

static int Main()
{
      Thread th1 = new Thread(ThreadMethod);
      Thread th2 = new Thread(ThreadMethod);

      sb = new StringBuilder("--------------------");
      th1.Start('1');
      th2.Start('2');
      th1.Join();
      th2.Join();
      Console.WriteLine(sb);

      int[] mas = new int[100];
      lock (mas.SyncRoot)
      {
            // безопасная обработка массива
      }

      return 0;
}
```

 Пример: Samples\5.2\5_2_3_lock.

Как правило, рекомендуется избегать блокировки членов типа **public** или экземпляров, которыми код не управляет. Например:

• **lock** (**this**) может привести к проблеме, если к экземпляру допускается открытый доступ извне потока;

• **lock** (**typeof** (MyType)) может привести к проблеме, если к MyType допускается открытый доступ извне потока;

• **lock** ("myLock") может привести к проблеме, поскольку любой код в процессе, используя ту же строку, будет совместно использовать ту же блокировку.

Поток может неоднократно блокировать один и тот же объект многократными вызовами метода Monitor.Enter или вложенными операторами **lock**. Объект будет освобожден, когда соответствующее количество раз будет вы-

зван метод Monitor.Exit или произойдет выход из самой внешней конструкции **lock**.

5.2.3.3. Синхронизация кода с помощью классов Mutex и Semaphore

Также синхронизацию кода можно выполнить, используя функциональность мьютексов (класс System.Threading.Mutex) и семафоров (класс System.Threading.Semaphore). Они работают медленнее, зато более универсальны – в частности, доступны из других процессов, поэтому блокировка осуществляется на уровне системы, а не отдельного процесса. Отличия этих двух классов в том, что мьютекс (от англ. mutual exclusion, взаимное исключение) гарантирует использование блокированного ресурса только одним потоком, а семафор позволяет нескольким потокам получить доступом к пулу ресурсов. Количество потоков, которые могут войти в семафор, ограничено. Счетчик на семафоре уменьшается на единицу каждый раз, когда в семафор входит поток, и увеличивается на единицу, когда поток освобождает семафор. Когда счетчик равен нулю, последующие запросы блокируются, пока другие потоки не освободят семафор. Когда семафор освобожден всеми потоками, счетчик имеет максимальное значение, заданное при создании семафора. В принципе, семафор со счетчиком, равным 1, соответствует мьютексу, только не имеет потока-хозяина.

Программисты Win32, которые занимались блокировкой ресурсов при написании программ на C++ (в частности, запрет запуска второй копии приложений и т.п.), должны помнить основы работы с мьютексами. В API Win32 это был не класс, а набор функций, поэтому создавались они вызовом функции CreateMutex, а удалялись – ReleaseMutex.

Перепишем предыдущий пример с использованием мьютекса. Дополнительно встроим защиту от повторного запуска программы:

```
static StringBuilder sb;
static Mutex mux;

static void ThreadMethod(object id)
{
      mux.WaitOne();
      for (int i = 0; i < sb.Length; i++)
      {
            sb[i] = (char)id;
            Thread.Sleep(100);
      }
      Console.WriteLine(sb);
      mux.ReleaseMutex();
```

```
	}

	static int Main()
	{
		Mutex mutex = new Mutex(false, "C#_Samples_5_2_3_mutex");

		if (!mutex.WaitOne(1000, false))
		{
			Console.WriteLine("В системе запущен другой экземпляр
программы!");
			return 1;
		}

		Thread th1 = new Thread(ThreadMethod);
		Thread th2 = new Thread(ThreadMethod);

		sb = new StringBuilder("--------------------");
		mux = new Mutex();
		th1.Start('1');
		th2.Start('2');
		th1.Join();
		th2.Join();
		Console.WriteLine(sb);
		return 0;
	}
```

 Пример: Samples\5.2\5_2_3_mutex.

Для демонстрации работы семафора перепишем класс, использующий пул потоков, чтобы одновременно запускалось не больше потоков, чем имеется процессорных ядер в системе:

```
	const int Counter = 10000;
	static Semaphore Sema;

	static void ThreadMethod(object id)
	{
		Sema.WaitOne();
		for (int i = 0; i < 15; i++)
		{
			for (int j = 0; j <= Counter; j++)
			{
				for (int k = 0; k <= Counter; k++)
				{
					if (j == Counter && k == Counter)
					{
						Console.Write(id);
						Thread.Sleep(100);
					}
				}
			}
		}
		Sema.Release();
	}

	static int Main()
	{
		int max_cpu, max_io, cur_cpu, cur_io;
```

```
            ThreadPool.GetMaxThreads(out max_cpu, out max_io);
            Sema = new Semaphore(Environment.ProcessorCount,
Environment.ProcessorCount);

            for (int i = 0; i < 4; i++)
            {
                  ThreadPool.QueueUserWorkItem(ThreadMethod, i + 1);
            }

            do
            {
                  Thread.Sleep(100);
                  ThreadPool.GetAvailableThreads(out cur_cpu, out cur_io);
            } while (cur_cpu != max_cpu);

            Console.WriteLine("\nВсе потоки завершили свою работу");
            return 0;
      }
```

Если семафор убрать, то потоки будут работать вперемешку. С семафором сначала отработает столько потоков, сколько имеется процессорных ядер в системе, а потом запустятся остальные. При необходимости количество потоков в пуле можно увеличить.

 Пример: Samples\5.2\5_2_3_semaphore.

5.2.3.4. Атомарные операции с классом Interlocked

Как уже было сказано выше, даже простейшие с точки зрения программиста операции (инкремент, декремент и т.д.) не являются атомарными с точки зрения потоков. А блокировка оператором **lock** или классом Monitor работает только для ссылочных объектов. Для того, чтобы сделать выполнение некоторых арифметических операций с числами атомарными, используется класс System.Threading.Interlocked (табл. 5.3).

Табл. 5.3 – Методы класса Interlocked

Метод	Описание
static int Add(**ref int** location, **int** value) **static long** Add(**ref long** location, **long** value)	Сложение двух чисел и помещение результата в первый аргумент как атомарная операция
static <тип> CompareExchange(ref <тип> location, <тип> value, <тип> comparand)	Если location = comparand, то произойдет замена value на location как атомарная операция(*)
static int Decrement(**ref int**	Уменьшение значения заданной пе-

location) **static long** Decrement(**ref long** location)	ременной и сохранение результата как атомарная операция
static <тип> Exchange(ref <тип> location, <тип> value)	Значение value заменяет location как атомарная операция(*)
static int Increment(**ref int** location) **static long** Increment(**ref long** location)	Увеличение значения заданной переменной и сохранение результата как атомарная операция
static long Read(**ref long** location)	Возвращает 64-битное значение, загруженное в виде атомарной операции

Примечание: (*) Есть версии для различных типов данных и универсальный метод.

5.2.3.5. Использование модификатора volatile

Ключевое слово **volatile** указывает, что поле может быть изменено несколькими потоками, выполняющимися одновременно. Поля, объявленные как **volatile**, не проходят оптимизацию компилятором, которая предусматривает доступ посредством отдельного потока. Это гарантирует наличие наиболее актуального значения в поле в любое время.

Как правило, модификатор **volatile** используется для поля, обращение к которому выполняется из нескольких потоков без использования оператора **lock**.

Ключевое слово **volatile** можно применять к полям следующих типов:

• ссылочным типам;

• типам указателей (в небезопасном контексте);

• типам **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **char**, **float** и **bool**;

• типу перечисления с одним из следующих базовых типов: **byte**, **sbyte**, **short**, **ushort**, **int** или **uint**;

• параметрам универсальных типов, являющихся ссылочными типами;

• IntPtr и UIntPtr.

Ключевое слово **volatile** можно применить только к полям класса или структуры. Локальные переменные не могут быть объявлены как **volatile**.

5.2.3.6. Потокобезопасность классов

Почти все типы .NET Framework, не являющиеся примитивными, также не являются и потокобезопасными, и все же они могут использоваться в многопоточном коде, если доступ к любому объекту защищен блокировкой.

Перечисление коллекций также не является потокобезопасной операцией, так как если другой поток меняет список в процессе перечисления, генерируется исключение. Однако, даже если бы коллекции были полностью потокобезопасными, это изменило бы немногое. Для примера рассмотрим добавление элемента к гипотетической потокобезопасной коллекции:

```
if (!myCollection.Contains(newItem) myCollection.Add(newItem);
```

Независимо от потокобезопасности собственно коллекции, данная конструкция в целом определенно не потокобезопасна. Заблокирован должен быть весь этот код целиком, чтобы предотвратить вытеснение потока между проверкой и добавлением нового элемента. Также блокировка должна быть использована везде, где изменяется список. К примеру, следующая конструкция должна быть обернута в блокировку для гарантии, что ее исполнение не будет прервано:

```
myCollection.Clear();
```

Другими словами, блокировки пришлось бы использовать точно так же, как с существующими потоконебезопасными классами.

Хуже всего дела обстоят со статическими полями с модификатором **public**. Для примера представим, что какое-либо статическое свойство структуры DateTime (например, DateTime.Now) потоконебезопасное, и два параллельных запроса могут привести к неправильным результатам или исключению. Единственная возможность исправить положение с использованием внешней блокировки – использовать конструкцию **lock** (**typeof** (DateTime)) при каждом обращении к DateTime.Now. Но мы не можем заставить делать это каждого программиста. Кроме того, как мы уже говорили, рекомендуется избегать блокировки типов. По этой причине статические поля структуры DateTime гарантированно потокобезопасны. Это обычное поведение типов в .NET Framework – статические члены потокобезопасны, нестатические – нет. Так же следует проектировать и наши собственные типы.

§ 5.3. Метаданные и отражение

Ранее мы говорили, что компилятор генерирует переносимый в Win32 исполняемый модуль (portable executable, PE), состоящий, главным образом, из MSIL-кода и *метаданных*. Метаданные – это данные в двоичном формате с описанием всех типов и их членов, которые определены в модуле или сборке. Одна из возможностей .NET позволяет вам писать код, чтобы обращаться к метаданным приложения посредством *отражения* (reflection). Отражение – это способность получать информацию о типах и их членах в период выполнения.

5.3.1. Иерархия API отражения

Программный интерфейс отражения в .NET представляет собой иерархию классов, определенную в пространстве имен System.Reflection:

```
System.Object
├── System.Reflection.Assembly
├── System.Reflection.MemberInfo
│   ├── System.Reflection.EventInfo
│   ├── System.Reflection.FieldInfo
│   ├── System.Reflection.MethodBase
│   │   ├── System.Reflection.ConstructorInfo
│   │   └── System.Reflection.MethodInfo
│   ├── System.Reflection.PropertyInfo
│   └── System.Type
└── System.Reflection.Module
```

Эти классы позволяют логически прослеживать информацию о сборках, модулях, типах и членах типа. Они очень функциональны, поэтому мы не будем перечислять все методы и поля каждого класса (это можно прочитать в справочной системе MSDN), а рассмотрим ряд примеров.

Центральное место в отражении занимает System.Type – абстрактный класс, который представляет тип в CTS (Common Type System) и позволяет запрашивать информацию о типе. Т.е. он представляет собой *метакласс*.

Для получения метакласса какого-либо типа можно использовать операцию **typeof** для типа или метод GetType для экземпляра этого типа:

```
int i = 123;
Console.WriteLine(i.GetType()); // System.Int32
Console.WriteLine(typeof(int)); // System.Int32
```

Также можно использовать статический метод GetType класса Type:

```
Console.WriteLine(Type.GetType("System.Int32"));
```

При вызове метода Type.GetType нельзя использовать псевдонимы языка C#, так как этот метод используется всеми языками .NET. Поэтому мы не можем указать применяемый в языке C# псевдоним **int** вместо System.Int32 или не указывать пространство имен.

 Пример: Samples\5.3\5_3_1_reflection.

5.3.1.1. Получение информации о типе

Класс System.Type также позволяет запрашивать практически все атрибуты типа, включая модификатор доступа, является ли тип вложенным, сведения о наследовании, атрибутах, параметрах универсального типа и т.д.:

```
      using System;
      using System.Collections;
      using System.Reflection;

      namespace ReflectionSample
      {
            class Program
            {
                  static void GetTypeInfo(Type t)
                  {
                        Type[] intfs = t.GetInterfaces();
                        Type[] gens = t.GetGenericArguments();

                        Console.WriteLine("Тип: {0}", t);
                        Console.WriteLine("Базовый тип: {0}", t.BaseType);
                        Console.Write("Реализуемые интерфейсы:");
                        foreach (Type intf in intfs) Console.Write(" {0}",
intf);
                        Console.WriteLine(intfs.Length > 0 ? "" : " нет");
                        Console.Write("Параметры типа:");
                        foreach (Type gen in gens) Console.Write(" {0}", gen);
                        Console.WriteLine(gens.Length > 0 ? "" : " нет");
                        Console.WriteLine("Абстрактный: {0}", t.IsAbstract);
                        Console.WriteLine("Массив: {0}", t.IsArray);
                        if (t.IsArray) Console.WriteLine("Размерность: {0}",
t.GetArrayRank());
                        Console.WriteLine("Класс: {0}", t.IsClass);
                        Console.WriteLine("Перечисление: {0}", t.IsEnum);
                        Console.WriteLine("Универсальный: {0}",
t.IsGenericType);
                        Console.WriteLine("Интерфейс: {0}", t.IsInterface);
                        Console.WriteLine("Вложенный: {0}", t.IsNested);
                        Console.WriteLine("Примитив: {0}", t.IsPrimitive);
                        Console.WriteLine("Открытый: {0}", t.IsPublic);
                        Console.WriteLine("Изолированный: {0}", t.IsSealed);
                        Console.WriteLine("Тип по зачению: {0}",
t.IsValueType);
                        Console.WriteLine("Модуль: {0}", t.Module);
                        Console.WriteLine("Пространство имен: {0}",
t.Namespace);
                        Console.WriteLine();
```

```
                }

                static int Main()
                {
                        GetTypeInfo(typeof(int));
                        GetTypeInfo(typeof(float[]));
                        GetTypeInfo(typeof(Type));
                        GetTypeInfo(typeof(IComparable<char>));
                        return 0;
                }
        }
}
```

Здесь рассмотрена только часть членов класса Type, определено же их гораздо больше. О запросе атрибутов типа мы поговорим в § 5.4.

5.3.1.2. Получение информации о членах типа

Если тип является структурой, классом или интерфейсом, можно получить полную информацию о его членах:

```
        static void GetMembersInfo(Type t)
        {
                Console.WriteLine("Тип: {0}", t);
                GetConstructorsInfo(t);
                GetMethodsInfo(t);
                GetFieldsInfo(t);
                GetPropertiesInfo(t);
                GetTypesInfo(t);
                Console.WriteLine();
        }

        static void GetConstructorsInfo(Type t)
        {
                ConstructorInfo[] ctors = t.GetConstructors();
                Console.WriteLine("  Конструкторы: {0}", ctors.Length > 0 ? "" :
"нет");

                foreach (var ctor in ctors)
                {
                        Console.WriteLine("    {0}", GetMethodInfo(ctor, t.Name));
                }
        }

        static void GetMethodsInfo(Type t)
        {
                MethodInfo[] meths = t.GetMethods();
                Console.WriteLine("  Методы: {0}", meths.Length > 0 ? "" : "нет");

                foreach (var met in meths)
                {
                        Console.WriteLine("    {0}", GetMethodInfo(met, met.Name));
                }
        }

        static string GetMethodInfo(MethodBase met, string name)
        {
```

```
            bool[] flags = { met.IsPublic, met.IsPrivate, met.IsStatic,
met.IsAbstract, met.IsVirtual};
            string[] mods = { "public", "private", "static", "abstract",
"virtual" };
            string rez;

            rez = GetModifiers(flags, mods);
            if (!met.IsConstructor) rez +=
GetTypeStr(((MethodInfo)met).ReturnParameter.ParameterType) + " ";
            rez += name + "(" + GetParamsInfo(met.GetParameters()) + ");";
            return rez;
      }

      static string GetModifiers(bool[] flags, string[] mods)
      {
            string rez = String.Empty;

            for (int i = 0; i < flags.Length; i++)
            {
                  if (flags[i]) rez += mods[i] + " ";
            }

            return rez;
      }

      static string GetParamsInfo(ParameterInfo[] pinfo)
      {
            string rez = String.Empty;

            foreach (var p in pinfo) rez += String.Format(p.Position > 0 ? ",
{0} {1}" : "{0} {1}",
                  GetTypeStr(p.ParameterType), p.Name);
            return rez;
      }

      static string GetTypeStr(Type t)
      {
            string type = t.ToString();

            int pos = type.LastIndexOf(".");
            if (pos >= 0) return type.Remove(0, pos + 1);
            return type;
      }

      static void GetFieldsInfo(Type t)
      {
            FieldInfo[] fields = t.GetFields();

            Console.WriteLine("  Поля: {0}", fields.Length > 0 ? "" : "нет");
            foreach (var field in fields)
            {
                  bool[] flags = { field.IsPublic, field.IsPrivate,
field.IsStatic, field.IsInitOnly, field.IsLiteral };
                  string[] mods = { "public", "private", "static", "readonly",
"const" };

                  Console.WriteLine("    {0}{1} {2};", GetModifiers(flags,
mods), GetTypeStr(field.FieldType), field.Name);
            }
      }
```

```
      static void GetPropertiesInfo(Type t)
      {
            PropertyInfo[] props = t.GetProperties();

            Console.WriteLine("  Свойства: {0}", props.Length > 0 ? "" :
"нет");
            foreach (var prop in props)
            {
                  ParameterInfo[] pinfo = prop.GetIndexParameters();

                  Console.Write("    {0} ", GetTypeStr(prop.PropertyType));
                  if (pinfo.Length > 0) Console.Write("this[{0}] {{",
GetParamsInfo(pinfo));
                  else Console.Write("{0} {{", prop.Name);
                  if (prop.CanRead) Console.Write(" {0}",
GetMethodInfo(prop.GetGetMethod(), "get"));
                  if (prop.CanWrite) Console.Write(" {0}",
GetMethodInfo(prop.GetSetMethod(), "set"));
                  Console.WriteLine(" }");
            }
      }

      static void GetTypesInfo(Type t)
      {
            Type[] types = t.GetNestedTypes();

            Console.WriteLine("  Вложенные типы: {0}", types.Length > 0 ? "" :
"нет");
            foreach (var type in types)
            {
                  Console.WriteLine("    {0}", type);
            }
      }
```

Теперь достаточно вызвать метод GetMembersInfo, чтобы для указанного типа получить информацию о его конструкторах, методах, полях, свойствах, индексаторах и вложенных типах. Например:

```
      GetMembersInfo(typeof(DateTime));
      GetMembersInfo(typeof(IEnumerator));
```

5.3.2. Работа со сборками и модулями

Сборка (assembly) – это физический файл, состоящий из нескольких PE-файлов .NET. Главное преимущество сборок заключается в том, что они позволяют семантически группировать функциональность, что облегчает развертывание приложения и управление его версиями. Представлением сборки в .NET является класс System.Reflection.Assembly. Класс Assembly позволяет выполнять множество действий, в том числе:

- перечисление модулей сборки;
- просмотр типов в сборке;

• определение идентификационной информации, такой, как имя и местоположение физического файла сборки;

• изучение информации о версиях и защите;

• получение точек входа сборки.

 Пример: Samples\5.3\5_3_2_assembly.

5.3.2.1. Просмотр типов сборки

Для просмотра всех типов, описанных в сборке нам нужно лишь создать экземпляр объекта Assembly и запросить массив Types для этой сборки, например:

```
      using System;
      using System.Diagnostics;
      using System.Reflection;
      using System.IO;

      namespace Test
      {
            public enum Enum1 { }
            internal enum Enum2 { }

            class AssemblyInfo
            {
                  public static void Get(string name)
                  {
                        Assembly a = Assembly.LoadFile(name);
                        Type[] types = a.GetTypes();

                        Console.WriteLine("Сборка: {0}", a.GetName());
                        Console.WriteLine("Типы:");
                        foreach (var type in types) Console.WriteLine("  {0}",
type);
                        Console.WriteLine();
                  }
            }
      }

      namespace AssemblySample
      {
            class Program
            {
                  static int Main()
                  {
                        string dir = Directory.GetCurrentDirectory();
                        string pname =
Process.GetCurrentProcess().ProcessName;

                        Test.AssemblyInfo.Get(dir + "\\" + pname + ".exe");
                        pname = dir +
"\\..\\..\\..\\5_3_1_reflection\\bin\\debug\\5_3_1_reflection.exe";
                        if (File.Exists(pname)) Test.AssemblyInfo.Get(pname);
            return 0;
```

```
                }
            }
        }
```

Для работы этого примера требуется наличие исполняемого файла предыдущего примера (5_3_1_reflection). В результате на консоли мы увидим информацию о типах текущей сборки и сборки 5_3_1_reflection:

```
      Сборка: 5_3_2_assembly, Version=1.0.3931.25329, Culture=neutral,
PublicKeyToken=null
      Типы:
        Test.Enum1
        Test.Enum2
        Test.AssemblyInfo
        AssemblySample.Program

      Сборка: 5_3_1_reflection, Version=1.0.3931.21376, Culture=neutral,
PublicKeyToken=null
      Типы:
        ReflectionSample.Program
```

Если программу запустить в режиме тестирования, то увидим следующую информацию о сборке:

```
      Сборка: vshost, Version=9.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
      Типы:
        Microsoft.VisualStudio.HostingProcess.EntryPoint
```

Так происходит потому, что сначала запускается отладчик, а уже затем он вызывает наше приложение.

5.3.2.2. Вывод списка модулей сборки

Все созданные нами ранее сборки состояли из одного модуля. Однако, к сборке можно подключать дополнительные модули. Это файлы, имеющие расширение .NETMODULE. К сожалению, создать такой файл из среды разработки нельзя (по крайней мере, настройки проекта в среде Visual Studio 2008 позволяют задать только три типа выходных данных – приложение Windows, консольное приложение и библиотека классов). Поэтому необходимо использовать командную строку компилятора:

```
      csc.exe /target:module ...
```

Создадим файл следующего содержания

```
      namespace MyModule
      {
            class MyClass { }
            struct MyStruct { }
            delegate void MyDelegate();
```

```
    }
```

и сохраним его под именем «ExtModule.cs». Далее откомпилируем его:

```
    csc.exe /target:module ExtModule.cs
```

Может потребоваться указание полного пути к компилятору C#. В каталоге примера находится пакетный файл «csext.bat», содержащий полную командную строку для .NET Framework версии 3.5. При необходимости номер версии можно исправить (см. п. 2.3.3). В итоге мы получили файл модуля «ExtModule.netmodule». Теперь его надо добавить в сборку, указав параметр компилятора

```
    csc.exe /addmodule:<имя модуля> ...
```

Например:

```
    csc.exe /out:5_3_2_assembly.exe /addmodule:ExtModule.netmodule
Program.cs Properties\AssemblyInfo.cs
```

В каталоге примера находится пакетный файл «csass.bat», содержащий полную командную строку для .NET Framework версии 3.5. Чтобы не набирать опции компилятора вручную, можно их скопировать из окна вывода данных о построении проекта (см. п. 4.1.3.2). Затем некоторые опции можно добавить, убрать или изменить.

После запуска файла «5_3_2_assembly.exe» из консоли видим результат:

```
    Состав текущей сборки:
      Модуль: 5_3_2_assembly.exe
      Типы:
        Test.Enum1
        Test.Enum2
        Test.AssemblyInfo
        AssemblySample.Program

      Модуль: ExtModule.netmodule
      Типы:
        MyModule.MyClass
        MyModule.MyStruct
        MyModule.MyDelegate
```

5.3.3. Позднее связывание и отражение

Позднее связывание (late binding) в программировании – это ситуация, когда компилятор ничего не знает о некоторых типах и их членах во время компоновки, но, тем не менее, специальные средства позволяют их использовать во время выполнения программы. В противовес этому ситуация, когда

информация о типе во время компиляции известна, называется *ранним связыванием* (early binding).

Примером позднего связывания может служить вызов динамических библиотек (DLL) в языке C++. Типы данных, описанные в них, использовать нельзя, но процедуры и функции – вполне. Еще один пример позднего связывания – использование COM-интерфейсов.

В среде .NET можно использовать динамические библиотеки, написанные на других языках. Это рассматривается как выполнение неуправляемого кода (см. § 5.5). Однако, если библиотека содержит управляемый код .NET, доступ к типам, описанным в такой библиотеке, можно получить посредством отражения (используя классы Assembly и System.Activator).

Сначала создадим проект «5_3_3_latebind». Поместим в файл «Program.cs» код следующего содержания:

```
using System;
using System.Reflection;
using System.IO;

namespace LateBindingSample
{
      class Program
      {
            static int Main()
            {
                  string[] dlls =
Directory.GetFiles(Directory.GetCurrentDirectory() + "\\..\\..\\", "*.dll");
                  string str = "Здравствуй, мир!";
                  int action;
                  ConsoleKeyInfo key;
                  object[] args;

                  foreach (string dll in dlls)
                  {
                        Assembly a = Assembly.LoadFrom(dll);
                        Type[] types = a.GetTypes();
                        // дальнейшие действия
                  }

                  Console.ReadKey(true);
                  return 0;
            }
      }
}
```

Итак, данный код перебирает все библиотеки в папке проекта (т.к. текущая папка при исполнении приложения – это <папка проекта>\bin\Debug или <папка проекта>\bin\Release, то, поднимаясь на два уровня вверх, попадаем в папку проекта). Затем загружает сборку из каждой найденной библио-

теки и извлекает из сборки все типы. Предположим, мы ищем типы для операций со строками. Тогда дальнейшие действия могут выглядеть так:

```
foreach (Type type in types)
{
      if (!type.IsAbstract)
      {
            MethodInfo mi1 = type.GetMethod("GetName");
            MethodInfo mi2 = type.GetMethod("GetOptions");
            MethodInfo mi3 = type.GetMethod("Convert");
            PropertyInfo pi = type.GetProperty("Data");

            if (mi1 != null && mi2 != null && mi3 != null && pi != null)
            {
                  object obj = Activator.CreateInstance(type);
                  string name = (string)mi1.Invoke(obj, null);
                  string[] opts = (string[])mi2.Invoke(obj, null);

                  Console.WriteLine("Найдена DLL: {0}", name);
                  Console.WriteLine("Выберите действие:");

                  for (int i = 0; i < opts.Length; i++)
                        Console.WriteLine("{0}) {1}", i + 1, opts[i]);

                  do
                  {
                        key = Console.ReadKey(true);
                        action = (int)key.KeyChar - 48;
                  } while (action < 1 || action > opts.Length);

                  Console.WriteLine(action);
                  args = new object[1] { str };
                  pi.GetSetMethod().Invoke(obj, args);
                  args = new object[1] { action - 1 };
                  Console.WriteLine(mi3.Invoke(obj, args));
            }
      }
}
```

Происходит поиск типов, имеющих методы GetName (чтобы узнать название операции), GetOptions (список доступных действий), Convert (выполнение операции) и свойство Data (доступ к инкапсулированной строке).

Далее создадим динамическую библиотеку. Для этого создадим новый проект, который назовем CaseConverter и сохраним его в папке проекта 5_3_3_latebind. Далее в опциях проекта («Проект» → «Свойства…») укажем:

• Тип выходных данных – «Библиотека классов» (на закладке «Приложение»);

• Путь вывода – «..\» (на закладке «Построение»), чтобы файл .DLL помещался не в папку 5_3_3_latebind\CaseConverter\obj\Debug или Release, как это установлено по умолчанию, а в папку 5_3_3_latebind (мы поднимаемся всего на один уровень вверх, т.к. для данного проекта текущей является пап-

ка CaseConverter).

В файл «Program.cs» данной библиотеки поместим следующий код:

```
using System;
using System.Text;

namespace StringConverters
{
      public class CaseConverter
      {
            StringBuilder SB = new StringBuilder("");
            static string[] Options = { "Верхний регистр", "Нижний
регистр", "Инвертировать регистр", "Не изменять регистр" };

            public string GetName()
            {
                  return "Преобразование регистра";
            }

            public string[] GetOptions()
            {
                  return Options;
            }

            public string Convert(string option)
            {
                  for (int i = 0; i < SB.Length; i++)
                  {
                        if (option == 0) SB[i] = char.ToUpper(SB[i]);
                        else if (option == 1) SB[i] =
char.ToLower(SB[i]);
                        else if (option == 2) SB[i] =
char.IsLower(SB[i]) ? char.ToUpper(SB[i]) : char.ToLower(SB[i]);
                        else if (option != 3) throw new
ArgumentException("Неизвестная опция");              }
                  return SB.ToString();
            }

            public string Data
            {
                  get { return SB.ToString(); }
                  set { SB = new StringBuilder(value); }
            }
      }

      class DLLEntryPoint
      {
            static int Main()
            {
                  return 0;
            }
      }
}
```

Данный класс умеет изменять регистр строки. Он содержит все необходимые методы и свойства, поэтому пройдет проверку в главной программе. Выполняем построение DLL (выполнить ее нельзя), затем запускаем

главный проект. Результаты работы:

```
Найдена DLL: Преобразование регистра
Выберите действие:
1) Верхний регистр
2) Нижний регистр
3) Инвертировать регистр
4) Не изменять регистр
3
зДРАВСТВУЙ, МИР!
```

 Пример: Samples\5.3\5_3_3_latebind.

5.3.4. Создание и исполнение кода в период выполнения

Мы уже знаем, как использовать отражение для работы с типами в период выполнения (раннее связывание), а также как осуществлять позднее связывание и динамическое исполнение кода. Теперь рассмотрим динамическое создание кода в период выполнения, т.е. создание кода «на лету». При динамическом создании типов и их членов в период выполнения используются классы пространства имен System.Reflection.Emit:

• Класс AssemblyBuilder (потомок Assembly) для динамического создания сборок;

• Класс ConstructorBuilder (потомок ConstructorInfo) для динамического создания конструкторов;

• Класс MethodBuilder (потомок MethodInfo) для динамического создания методов;

• Класс FieldBuilder (потомок FieldInfo) для динамического создания полей;

• и многие другие.

С помощью классов из этого пространства имен можно создавать сборку в памяти, добавлять в нее модули, определять для модулей новые типы (и их члены) и даже генерировать код MSIL для реализации методов.

Напишем программу, динамически создающую класс со статическим методом, выводящим на консоль строку «Hello, World!»:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace EmittingSample
{
      class Program
      {
```

```
                  static int Main()
                  {
                        Console.WriteLine("Создаем динамический класс
EmitClass...");
                        AppDomain domain = AppDomain.CurrentDomain;
                        AssemblyName aname = new AssemblyName("EmitAssembly");
                        AssemblyBuilder abuild =
domain.DefineDynamicAssembly(aname, AssemblyBuilderAccess.Run);
                        ModuleBuilder modbuild =
abuild.DefineDynamicModule("EmitModule");
                        TypeBuilder tbuild = modbuild.DefineType("EmitClass",
TypeAttributes.Public);
                        MethodBuilder mbuild =
tbuild.DefineMethod("HelloWorld1", MethodAttributes.Public |
MethodAttributes.Static);
                        ILGenerator msil = mbuild.GetILGenerator();
                        msil.EmitWriteLine("Hello, World!");
                        msil.Emit(OpCodes.Ret);
                        mbuild = tbuild.DefineMethod("HelloWorld2",
MethodAttributes.Public);
                        msil = mbuild.GetILGenerator();
                        msil.EmitWriteLine("Hello, World!");
                        msil.Emit(OpCodes.Ret);

                        Console.WriteLine("Создаем экземпляр класса
EmitClass...");
                        Type etype = tbuild.CreateType();
                        object eclass = Activator.CreateInstance(etype);

                        Console.WriteLine("Получаем информацию о методах
HelloWorld...");
                        MethodInfo hello1 = etype.GetMethod("HelloWorld1");
                        MethodInfo hello2 = etype.GetMethod("HelloWorld2");

                        Console.WriteLine("Вызываем статический метод
HelloWorld...");
                        hello1.Invoke(null, null);
                        Console.WriteLine("Вызываем экземплярный метод
HelloWorld...");
                        hello1.Invoke(eclass, null);
                        return 0;
                  }
            }
      }
```

Вывод на консоль:

```
Создаем динамический класс EmitClass...
Создаем экземпляр класса EmitClass...
Получаем информацию о методах HelloWorld...
Вызываем статический метод HelloWorld...
Hello, World!
Вызываем экземплярный метод HelloWorld...
Hello, World!
```

Сначала мы получаем ссылку на текущий домен приложения. Затем создаем экземпляр класса AssemblyName, описывающего сборку (с его помощью можно задать имя сборки, номер версии, региональные параметры и

т.д.). Назовем сборку «EmitAssembly». Затем в текущем домене создаем динамическую сборку – экземпляр класса AssemblyBuilder, используя сформированное ранее описание сборки и указывая, что сборка будет только запускаться на исполнение (Run). В перечислении AssemblyBuilderAccess есть и другие константы, позволяющие записать сформированный код MSIL в файл и т.п. Затем в динамической сборке создаем новый динамический модуль «EmitModule» (экземпляр класса ModuleBuilder), а в нем – новый динамический тип «EmitClass» (экземпляр класса TypeBuilder) с модификатором **public**, чтобы он был виден за пределами сборки EmitAssembly. Далее в этом класса создаем статический метод HelloWorld1 и метод экземпляра HelloWorld2 (экземпляры класса MethodBuilder), оба с модификатором **public**. Класс ILGenerator позволяет добавлять код в динамические методы, включающий операторы выражений и операторы языка C# (метод Emit). Для некоторых операций и выражений есть специальные методы, например, метод EmitWriteLine для добавления в динамический код вызова метода Console.WriteLine. В данном случае в оба метода помещается код для вывода на консоль строки «Hello, World!» и оператор завершения метода **return** (он требуется обязательно, даже если метод ничего не возвращает).

Далее используем конструктор типов для создания метакласса описанного типа. После чего уже знакомый нам класс Activator используется для создания экземпляра класса EmitClass (он нам потребуется для вызова метода HelloWorld2). Затем также уже знакомый метод GetMethod получает информацию о методах. Далее мы эти методы вызываем, указывая ссылку на экземпляр класса для метода экземпляра. Параметров у методов нет, поэтому второй аргумент метода Invoke – **null**.

 Пример: Samples\5.3\5_3_4_emit.

§ 5.4. Атрибуты

Большинство языков программирования разрабатываются с учетом минимального набора необходимых возможностей. Например, мы уже изучили, какие модификаторы могут иметь классы, интерфейсы, методы, делегаты и т.д. Между тем, если ограничиться только ими, то можно отметить два недостатка:

1) Невозможно предвидеть все усовершенствования, которые могут потребоваться в будущем, и как они повлияют на способы выражения типов на данном языке;

2) Так как компиляторы распознают только предопределенные ключевые слова, отсутствует возможность создавать свои собственные.

Скажем, как создать связь между классом, написанным на языке C++, и URL-ссылкой на документацию для данного класса? Или как мы будете ассоциировать члены классов полями XML-файла или системой справки? Поскольку язык C++ разрабатывался задолго до прихода в нашу жизнь URL и XML, обе эти задачи выполнить довольно трудно.

До сих пор решения подобных проблем предполагают хранение дополнительной информации в отдельном файле (DEF, IDL и т.д.), которая затем связывается с тем или иным типом или членом. Главная проблема в том, что класс больше не является «самоописывающимся», т.е. теперь пользователь не может сказать о классе все, лишь взглянув на его определение.

Язык C# предлагает иную парадигму – *атрибуты* (attributes). Атрибуты предоставляют универсальные средства связи данных с типами. Поскольку мы можем создавать атрибуты на основе любой информации, существует стандартный механизм определения самих атрибутов и запроса членов или типов в период выполнения как связанных с ними атрибутов.

5.4.1. Синтаксис описания атрибутов

Синтаксис описания атрибутов следующий:

```
<атрибуты> :: "[" [<целевой объект> :] <атрибут1>, [<атрибут2>, ...] "]"

<целевой объект> :: assembly
<целевой объект> :: event
<целевой объект> :: field
<целевой объект> :: method
<целевой объект> :: module
```

```
<целевой объект> :: param
<целевой объект> :: property
<целевой объект> :: return
<целевой объект> :: type

<атрибут> :: <тип атрибута> [(<аргументы>)]

<аргументы> :: <позиционные аргументы> [, <именованные аргументы>]
<аргументы> :: <именованные аргументы>

<позиционные аргументы> :: <список фактических аргументов>
<именованные аргументы> :: <идентификатор> = <выражение> [, ...]
```

Целевой объект указывается, когда контекст применения атрибута неоднозначен. Например:

```
[SomeAttribute] public int F (void);
```

Здесь атрибут может относиться как к методу, так и к возвращаемому им значению. В этом случае компилятор выбирает наиболее употребительный вариант:

1. Глобальный атрибут может применяться либо к сборке, либо к модулю. Для этого контекста нет значения по умолчанию, поэтому обязательно требуется указать целевой объект **assembly** или **module** соответственно;

2. Атрибут делегата может применяться либо к объявляемому делегату (по умолчанию или при указании **type**), либо к его возвращаемому значению (**return**);

3. Атрибут метода может применяться либо к объявляемому методу (по умолчанию или при указании **method**), либо к его возвращаемому значению (**return**). Здесь под методами также подразумеваются операторы и методы доступа **get** при объявлении свойства или индексатора;

4. Атрибут события может применяться к объявляемому событию (по умолчанию или указанию **event**), к связанному полю (**field**) или к связанным методам **add** и **remove** (**method**). Если событие имеет методы доступа, допустимо только указание **event**;

5. Атрибут метода доступа **set** при объявлении свойства или индексатора, а также атрибут метода доступа **add** и **remove** при объявлении события может применяться либо к связанному методу (по умолчанию или при указании **method**), либо к его одиночному неявному параметру **value** (**param**), либо к возвращаемому значению (**return**).

В других контекстах описание целевого объекта разрешено, но излишне, т.к. доступен всего один объект, и он используется по умолчанию:

6. При объявлении перечислений, структур, классов и интерфейсов можно или включать описатель **type**, или опускать его;

7. При объявлении констант и полей можно или включать описатель **field**, или опускать его;

8. При объявлении конструкторов и деструкторов можно или включать описатель **method**, или опускать его;

9. При объявлении свойства или индексатора можно или включать описатель **property**, или опускать его;

10. При описании формального параметра можно или включать описатель **param**, или опускать его.

Задание недопустимого целевого объекта, например, **param** в объявлении класса, влечет предупреждение от компилятора. Такой атрибут будет проигнорирован. Использование атрибута с недопустимым объектом (например, при описании пространства имен) влечет ошибку компиляции.

Тип атрибута – имя класса, являющегося потомком System.Attribute. В принципе, здесь происходит создание экземпляра указанного класса. Позиционные аргументы – это фактические параметры конструктора, а именованные аргументы – список инициализации свойств класса. С аргументами мы разберемся позже, а пока опишем свой класс-потомок Attribute (базовый класс Attribute использовать нельзя, т.к. он абстрактный), и рассмотрим варианты его применения:

```
class MyAttr : Attribute {}

[module: MyAttribyte] public delegate int Del1(int i); // Предупреждение
[type: MyAttr] public delegate int Del2(int i);
[return: MyAttr] public delegate int Del3(int i);

[type: MyAttr] enum MyEnum { }
[type: MyAttr] struct MyStruct { };
[type: MyAttr] interface IMyInterface { }

[type: MyAttr] class MyClass
{
      [field: MyAttr] public const int X = 5;
      [field: MyAttr] public int Y = 5;

      [method: MyAttr] public MyClass() { }
      [method: MyAttr] ~MyClass() { }
      public void F1([param: MyAttr] int i) { }
      [method: MyAttr][return: MyAttr] public void F2() { }

      [property: MyAttr] public int Prop
      {
            [method: MyAttr][return: MyAttr] get { return Y; }
            [return: MyAttr][param: MyAttr] set { Y = value; }
```

```
        }

        [event: MyAttr][field: MyAttr] public event Del2 Event1;
        [event: MyAttr] public event Del2 Event2
        {
              [return: MyAttr][param: MyAttr] add { }
              [return: MyAttr][method: MyAttr] remove { }
        }
}
```

При объявлении собственного класса атрибута рекомендуется в его имени использовать окончание Attribute. При использовании атрибута это окончание можно опускать, оно добавляется компилятором автоматически. Если возникает неоднозначность, тип атрибута предваряется символом «@». В этом случае окончание добавляться не будет:

```
class X : Attribute { }
class XAttribute : Attribute { }
class Y : Attribute { }
class ZAttribute : Attribute { }

[X] class MyClass1 {} // Ошибка
[@X] class MyClass2 {} // OK
[XAttribute] class MyClass3 { } // OK
[@XAttribute] class MyClass4 { } // OK
[Y] class MyClass5 { } // OK
[Z] class MyClass6 { } // OK
[ZAttribute] class MyClass7 { } // OK
```

 Пример: Samples\5.4\5_4_1_attr.

5.4.2. Определение и запрос атрибутов

5.4.2.1. Параметры атрибута

Итак, рассмотрим, чем отличаются позиционные и именованные аргументы атрибутов. Позиционные аргументы – это фактические параметры конструктора класса атрибута. Именованные аргументы – это записи вида

```
<идентификатор> = <выражение>
```

где идентификатор – это имя нестатического поля или свойства класса атрибута, доступного как для чтения, так и для записи.

Позиционные аргументы должны стоять в начале списка аргументов, и соответствовать порядку формальных аргументов конструктора. После них могут располагаться в произвольном порядке именованные аргументы.

Типы позиционных и именованных аргументов атрибута ограничены следующим набором:

• Все типы по значению, кроме **decimal**, структур и обнуляемых типов. Перечисления допускаются при условии, что они и все типы, по отношению к которым они являются вложенными, открытые (**public**);

• System.Type;

• **object**;

• одномерный массив значений любого из вышеуказанных типов.

Пример:

```
public class MyAttr : Attribute
{
      public MyAttr(int i) { }
      public MyAttr(int i, int j) { }
      public const int X = 1;
      public readonly int Y = 2;
      public int Z = 3;
      public decimal U = 4;
      public int A { get { return 0; } }
      public int B { set { } }
      public int C { get; set; }
      public double D { get; set; }
      public string Str { get; set; }
}

[MyAttr] public enum E1 { } // Ошибка
[MyAttr("x", B = 7)] public enum E2 { } // Ошибка
[MyAttr("x", X = 5, Y = 5)] public enum E3 { } // Ошибка
[MyAttr(5, 7, A = 1)] public enum E4 { } // Ошибка
[MyAttr("x", C = 5, D = 0.5, Str = "ABC")] public enum E5 { } // OK
[MyAttr(5, 7, Str = "Привет!", Z = 7)] public enum E6 { } // OK
[MyAttr("x", U = 1, Str = "Привет!", Z = 7)] public enum E7 { } //
Ошибка
```

В данном примере класс атрибута имеет два конструктора. Поэтому при описании атрибута должны быть обязательно указаны позиционные аргументы – один параметр типа **string** или два параметра типа **int**. В качестве именованных аргументов можно в произвольном порядке указывать Z, C, D, Str. Они имеют допустимый тип и доступ для чтения и записи.

 Пример: Samples\5.4\5_4_2_params.

Давайте рассмотрим более конкретный вариант определения атрибутов. Предположим, мы хотим для каждого поля некоторого класса определить его формат и цвет текста и фона при выводе на консоль. Опишем следующие классы атрибутов:

```
class FormattableAttribute : Attribute
{
}

class FormatAttribute : Attribute
{
```

```
            private string faFormat;
            private ConsoleColor faColor = ConsoleColor.Gray;
            private ConsoleColor faBackground = ConsoleColor.Black;

            public FormatAttribute(string format)
            {
                  faFormat = format;
            }

            public string Format
            {
                  get { return faFormat; }
            }

            public ConsoleColor Color
            {
                  get { return faColor; }
                  set { faColor = value; }
            }

            public ConsoleColor Background
            {
                  get { return faBackground; }
                  set { faBackground = value; }
            }
      }
```

Первый класс атрибута не содержит параметров, и будет указываться лишь для того, чтобы показать, что тип имеет атрибуты форматирования полей. Второй класс будет использоваться для указания формата вывода полей типа. Здесь требуется обязательное указание формата, а указание цвета символов и фона опционально (по умолчанию – светло-серый и черный соответственно):

```
      [Formattable] class MyClass
      {
            [Format("{0:P2}", Color = ConsoleColor.Blue)]
            public double Per = 0.7581;
            [Format("{0}", Background = ConsoleColor.Red)]
            public string Str = "Hello!";
            [Format("0x{0,4:X4}", Color = ConsoleColor.Yellow, Background =
ConsoleColor.Blue)]
            public int Val = 123;
      }
```

5.4.2.2. Запрос атрибутов

Для запроса типа или члена о прикрепленных к ним атрибутах применяется отражение. Способ получения атрибута зависит от типа члена, к которому производится запрос:

1. Метод GetCustomAttributes класса Type возвращает атрибуты типа (все или только атрибуты указанного класса атрибутов).

2. Метод GetCustomAttributes класса MemberInfo возвращает атрибуты члена типа. Этот метод перегружен для различных видов членов типа в классах-потомках MemberInfo (см. п. 5.3.1).

3. Методы GetCustomAttribute и GetCustomAttributes класса Attribute возвращают атрибуты сборки, модуля, типа или члена типа.

У первых двух методов две реализации:

```
object[] GetCustomAttributes(bool inherit);
object[] GetCustomAttributes(Type attributeType, bool inherit);
```

Первая реализация возвращает массив всех атрибутов типа или члена, вторая – только атрибутов указанного класса (или производных от него классов атрибутов). Логический параметр определяет, следует ли дополнительно искать атрибуты у базовых типов (для метода Type.GetCustomAttributes) или у членов базовых типов, которые переопределяются данным членом (для метода MemberInfo.GetCustomAttributes). У методов класса Attribute появляется дополнительные параметры – ссылки на метаклассы сборки, модуля, типа или члена типа.

Пример:

```
static void GetAttribute(object x)
{
      FormattableAttribute fattr =
(FormattableAttribute)Attribute.GetCustomAttribute(x.GetType(),
typeof(FormattableAttribute));

      if (fattr != null)
      {
            FieldInfo[] fields = x.GetType().GetFields();

            foreach (FieldInfo field in fields)
            {
                  FormatAttribute attr =
(FormatAttribute)Attribute.GetCustomAttribute(field,
typeof(FormatAttribute));

                  if (attr != null)
                  {
                        Console.WriteLine("Формат: {0}", attr.Format);
                        Console.WriteLine("Цвет: {0}", attr.Color);
                        Console.WriteLine("Фон: {0}", attr.Background);
                        Console.BackgroundColor = attr.Background;
                        Console.ForegroundColor = attr.Color;
                        Console.WriteLine(String.Format(attr.Format,
field.GetValue(x)));
                        Console.ResetColor();
                  }
            }
      }
}
```

В данном примере сначала мы проверяем, что тип класса «x» описан с атрибутом FormattableAttribute. Если это так, то, используя отражение, перебираем все поля этого класса и ищем среди них те, что описаны с атрибутом FormatAttribute. После этого используем параметры атрибута для вывода на консоль значения поля. Так, при следующем вызове данного метода

```
static int Main()
{
      MyClass cls = new MyClass();

      GetAttribute(cls);
      return 0;
}
```

вывод на консоль будет следующим:

```
Формат: {0:P2}
Цвет: Blue
Фон: Black
75,81% // синий шрифт
Формат: {0}
Цвет: Gray
Фон: Red
Hello! // красный фон, серый шрифт
Формат: 0x{0,4:X4}
Цвет: Yellow
Фон: Blue
0x007B // синий фон, желтый шрифт
```

 Пример: Samples\5.4\5_4_2_request.

5.4.3. Атрибут AttributeUsage

С помощью атрибута AttributeUsage, если его использовать при описании класса атрибута, можно определить допустимые способы его применения. Синтаксис:

```
"["AttributeUsage(<значение> [, AllowMultiple = <значение>] [, Inherited
= <значение>])"]"
```

 Пример: Samples\5.4\5_4_3_usage.

5.4.3.1. Указание целевых объектов атрибута

Первый параметр является позиционным. Он позволяет задавать целевые объекты, к которым может быть прикреплен атрибут. Тип этого параметра – перечисление AttributeTargets (табл. 5.4).

Табл. 5.4 – Константы перечисления AttributeTargets

Константа	Значение	Описание
Assembly	0x0001	Атрибут применим к сборке
Module	0x0002	Атрибут применим к модулю
Class	0x0004	Атрибут применим к классу
Struct	0x0008	Атрибут применим к структуре
Enum	0x0010	Атрибут применим к перечислению
Constructor	0x0020	Атрибут применим к конструктору
Method	0x0040	Атрибут применим к методу
Property	0x0080	Атрибут применим к свойству или индексатору
Field	0x0100	Атрибут применим к полю
Event	0x0200	Атрибут применим к событию
Interface	0x0400	Атрибут применим к интерфейсу
Parameter	0x0800	Атрибут применим к параметру метода
Delegate	0x1000	Атрибут применим к делегату
ReturnValue	0x2000	Атрибут применим к возвращаемому значению
GenericParameter	0x4000	Атрибут применим к универсальному параметру
All	0x7fff	Атрибут применим к любому элементу программы

Несколько значений можно объединять операцией дизъюнкции. Укажем для классов атрибутов, описанных нами выше, что атрибут FormattableAttribute может применяться только к классам, а атрибут FormatAttribute – к полям класса:

```
[AttributeUsage(AttributeTargets.Class)]
class FormattableAttribute : Attribute
// ...

[AttributeUsage(AttributeTargets.Field)]
class FormatAttribute : Attribute
// ...
```

Теперь попытка использовать атрибут не по назначению будет приводить к ошибке компиляции:

```
[Format("G")] class MyClass2 { } // Ошибка
```

5.4.3.2. Атрибуты однократного и многократного использования

Второй параметр разрешает (если его значение равно **true**) или запрещает (**false**, по умолчанию) многократное применение атрибута. Например, разрешим задание для поля сразу нескольких атрибутов FormatAttribute:

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = true)]
class FormatAttribute : Attribute
// ...
```

Теперь его можно применять несколько раз к одному и тому же полю, а атрибут FormattableAttribute может быть только единожды применен к одному классу:

```
[Formattable] class MyClass
{
      [Format("{0:P2}", Color = ConsoleColor.Blue)]
      [Format("{0:E}", Color = ConsoleColor.Green)]
      public double Per = 0.7581;
      // ...
}

[Formattable, Formattable] class MyClass3 { } // Ошибка
```

Теперь для запроса атрибутов нельзя применять использованный нами ранее метод Attribute.GetCustomAttribute – он возвращает ссылку на атрибут только в том случае, если он один. Иначе возникает исключение

```
    Необработанное исключение: System.Reflection.AmbiguousMatchException:
Обнаружено несколько пользовательских атрибутов одного типа.
```

Поэтому используем метод MemberInfo.GetCustomAttributes, который возвращает массив атрибутов:

```
    FormatAttribute[] attrs =
(FormatAttribute[])field.GetCustomAttributes(typeof(FormatAttribute), false);

    foreach(var attr in attrs)
    {
          Console.WriteLine("Формат: {0}", attr.Format);
          Console.WriteLine("Цвет: {0}", attr.Color);
          Console.WriteLine("Фон: {0}", attr.Background);
          Console.BackgroundColor = attr.Background;
          Console.ForegroundColor = attr.Color;
          Console.WriteLine(String.Format(attr.Format, field.GetValue(x)));
          Console.ResetColor();
    }
```

Результат работы программы:

```
Формат: {0:E}
Цвет: Green
Фон: Black
7,581000E-001 // ярко-зеленый шрифт
```

```
      Формат: {0:P2}
      Цвет: Blue
      Фон: Black
      75,81% // синий шрифт
      Формат: {0}
      Цвет: Gray
      Фон: Red
      Hello! // красный фон, серый шрифт
      Формат: 0x{0,4:X4}
      Цвет: Yellow
      Фон: Blue
      0x007B // синий фон, желтый шрифт
```

5.4.3.3. Задание правил наследования атрибутов

Последний параметр – флаг Inherited – определяет, может ли атрибут наследоваться производными классами и переопределенными членами. Его значение по умолчанию равно **true.** Если флаг установить в **false**, атрибут не будет наследоваться, в противном случае его действие зависит от значения флага AllowMultiple. Если Inherited установлен в **true**, а AllowMultiple – в **false**, установленный атрибут заменит унаследованный. Однако если и Inherited, и AllowMultiple установлены в **true**, атрибуты члена будут аккумулироваться.

Пример:

```
      class TestAttribute : Attribute { public string Value; }
      [AttributeUsage(AttributeTargets.Class, Inherited = false)]
      class TestAttribute1 : TestAttribute { }
      [AttributeUsage(AttributeTargets.Class, Inherited = true)]
      class TestAttribute2 : TestAttribute { }
      [AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited
= true)]
      class TestAttribute3 : TestAttribute { }

      [TestAttribute1(Value = "Base1")] class BaseClass1 { }
      [TestAttribute2(Value = "Base2")] class BaseClass2 { }
      [TestAttribute3(Value = "Base3")] class BaseClass3 { }

      class ChildClass1 : BaseClass1 { }
      class ChildClass21 : BaseClass2 { }
      [TestAttribute2(Value = "Child2")] class ChildClass22 : BaseClass2 { }
      [TestAttribute3(Value = "Child3")] class ChildClass3 : BaseClass3 { }

      static void GetAttributes(Type type)
      {
            TestAttribute[] attrs =
(TestAttribute[])type.GetCustomAttributes(typeof(TestAttribute), true);

            Console.WriteLine("Тип {0} имеет следующие атрибуты:", type.Name);
            foreach (var attr in attrs) Console.WriteLine("  {0} (Value =
{1})", attr.GetType().Name, attr.Value);
            Console.WriteLine();
      }
```

```
static int Main()
{
      GetAttributes(typeof(ChildClass1));
      GetAttributes(typeof(ChildClass21));
      GetAttributes(typeof(ChildClass22));
      GetAttributes(typeof(ChildClass3));
      return 0;
}
```

Обратите внимание, что при вызове метода GetCustomAttributes мы указали параметр inherit = **true**, иначе получили бы только атрибуты дочерних классов. Результат работы программы:

```
Тип ChildClass1 имеет следующие атрибуты:

Тип ChildClass21 имеет следующие атрибуты:
  TestAttribute2 (Value = Base2)

Тип ChildClass22 имеет следующие атрибуты:
  TestAttribute2 (Value = Child2)

Тип ChildClass3 имеет следующие атрибуты:
  TestAttribute3 (Value = Child3)
  TestAttribute3 (Value = Base3)
```

5.4.4. Стандартные классы атрибутов

В справочной системе MSDN (на странице класса Attribute) приведена иерархия большинства из имеющихся классов атрибутов. Это, например:

• Уже знакомый нам атрибут System.AttributeUsageAttribute;

• Атрибут System.Diagnostics.ConditionalAttribute, используемый для определения условных методов;

• Атрибут System.ObsoleteAttribute, используемый для пометки типа или члена как устаревшего;

• Атрибут System.FlagsAttribute, указывающий, что перечисление может обрабатываться как битовое поле, которое является набором флагов (см. п. 3.1.2.5);

• Атрибут System.Runtime.InteropServices.DllImportAttribute для вызова функций, экспортированных из неуправляемых динамических библиотек (DLL) (см. § 5.5);

• Ряд атрибутов пространства имен System.Reflection для указания параметров сборки (автоматически задаются в файле AssemblyInfo.cs);

• и т.д.

Первые три имени атрибутов являются зарезервированными и не могут

использоваться при создании новых классов атрибутов.

 Пример: Samples\5.4\5_4_4_stdattr.

5.4.4.1. Объявление условных методов

Условные методы похожи на методы, объявленные с директивами условной компиляции, но есть одно отличие:

```
#define X
using System;
using System.Diagnostics;

namespace StandardAttributesSample
{
      class Program
      {
#if X
            static void Method1()
            {
                  Console.WriteLine("Метод 1");
            }
#endif
#if Y
            static void Method2()
            {
                  Console.WriteLine("Метод 2");
            }
#endif
            [Conditional("X")] static void Method3()
            {
                  Console.WriteLine("Метод 3");
            }
            [Conditional("Y")] static void Method4()
            {
                  Console.WriteLine("Метод 4");
            }
            [Conditional("Y"), Conditional("X")] static void Method5()
            {
                  Console.WriteLine("Метод 5");
            }

            static int Main()
            {
                  Method1(); // OK
                  Method2(); // Ошибка
                  Method3(); // OK
                  Method4(); // OK
                  Method5(); // OK
                  return 0;
            }
      }
}
```

Метод, заключенный в директиву условной компиляции, исключается из кода программы и не может быть вызван, если символ условной компиля-

ции не определен. Условный метод, описанный с атрибутом Conditional("<символ>"), где <символ> – это символ условной компиляции, может быть вызван в любом случае. Но его тело будет выполнено только в том случае, если этот символ определен:

```
Метод 1
Метод 3
Метод 5
```

Если указано несколько атрибутов Conditional, то метод будет выполнен, если определен хотя бы один из указанных символов (т.е. действует операция дизъюнкции).

На условные методы налагается ряд ограничений – это должны быть методы класса или структуры, имеющие тип возвращаемого значения **void**, не имеющие модификатора **override** и не являющиеся реализацией метода интерфейса.

5.4.4.2. Пометка типа или члена как устаревшего

Атрибут Obsolete помечает тип или член типа как устаревший. Позиционный параметр соответствует сообщению, которое получит пользователь от компилятора при попытке использовать такой тип или член:

```
[Obsolete("Класс Z устарел. Используйте вместо него класс NewZ")]
class Z { }
class NewZ { }
class ZZ
{
      [Obsolete("Метод F1 устарел и будет исключен из класса в следующей
версии библиотеки. Используйте вместо него метод F2")]
      public static void F1() { }
      public static void F2() { }
}

static int Main()
{
      Z z = new Z(); // Предупреждение
      ZZ.F1(); // Предупреждение
      return 0;
}
```

§ 5.5. Неуправляемый код

Рассмотрим три основных варианта применения неуправляемого кода в .NET:

• Службы Platform Invocation Services, позволяющие коду .NET обращаться к неуправляемым библиотекам DLL;

• Написание блоков небезопасного кода (unsafe code), позволяющее использовать в приложениях такие конструкции, как указатели;

• Организация взаимодействия с COM (COM interoperability). Данная тема не входит в список рассматриваемых вопросов. В принципе, технология COM, хотя пока еще достаточно широко используется, считается устаревшей (т.к. была разработана корпорацией Microsoft еще в 1993 году). На сегодняшний день Microsoft объявила рекомендуемой основой для создания приложений и компонентов под Windows платформу .NET.

5.5.1. Службы Platform Invocation Services

Службы Platform Invocation Services .NET (или PInvoke) позволяют управляемому коду работать с функциями и структурами, экспортированными из DLL. Мы можем создавать DLL с неуправляемым кодом с помощью компиляторов .NET (неуправляемые DLL), либо с помощью других языков программирования.

Поскольку компилятор не имеет доступа к исходному коду DLL, мы должны указать ему сигнатуру встроенного метода, информацию о любых возвращаемых значениях, а также способы преобразования параметров для DLL.

 Пример: Samples\5.5\5_5_1_pinvoke.

5.5.1.1. Объявление импортируемой функции

Рассмотрим объявление на языке C# импорта функций из динамических библиотек с неуправляемым кодом. Для этого используется атрибут System.Runtime.InteropServices.DllImportAttribute. Общий синтаксис следующий:

```
    "["DLLImport("<имя DLL>" [, <именованные параметры>])"]" [<модификатор доступа>] static extern <тип возвращаемого значения> <имя метода>(<список формальных аргументов>);
```

Перечислим некоторые именованные параметры:

• **bool** BestFitMapping – включает или отключает поведение наилучшего сопоставления при преобразовании знаков Unicode в знаки ANSI (по умолчанию **true**).

• CharSet CharSet – задает кодировку строк при вызове импортируемого метода (доступны константы Ansi – по умолчанию, Unicode, Auto).

• **bool** ExactSpelling – определяет, требуется ли точное совпадение имени импортируемой функции, или будет происходить поиск отдельных версий для кодировок ANSI и Unicode (по умолчанию **false**).

Почему так много внимания уделяется кодировке строк? Все функции API Windows, параметрами которых являются строки, имеют две версии – для кодировок ANSI (1 байт на символ, имена таких функций заканчиваются на A) и для кодировок Unicode (заканчиваются на W). Например, смотрим определение функции MessageBox (см. заголовочный файл winuser.h):

```
WINUSERAPI int WINAPI MessageBoxA(
    HWND hWnd ,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
WINUSERAPI int WINAPI MessageBoxW(
    HWND hWnd ,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
#ifdef UNICODE
#define MessageBox  MessageBoxW
#else
#define MessageBox  MessageBoxA
#endif // !UNICODE
```

Здесь второй параметр – текст сообщения, третий – заголовок окна сообщения. Поэтому перед службой PInvoke стоит два вопроса при вызове импортируемой функции. Во-первых, искать ли в DLL функцию точно с таким именем, как это указано, либо добавлять суффикс A или W. Во-вторых, нужно ли преобразовывать строковые параметры из кодировки Unicode (которую имеют все строки в языке C#) в кодировку ANSI (какая это именно будет кодировка – зависит от языка локализации Windows).

Типы остальных параметров надо подбирать так, чтобы у них совпадал размер в байтах. Например, первый параметр типа HWND – указатель на окно, показывающее сообщение. Любой указатель в Win32 занимает 4 байта, поэтому при описании импортируемой функции можно использовать тип **int** или **uint**.

• CallingConvention CallingConvention – указывает соглашение о вызове импортируемой функции. Доступны константы Winapi (по умолчанию), Cdecl, StdCall, ThisCall, FastCall.

В языке C++ есть несколько модификаторов для указания соглашения о вызове **(__cdecl**, **__stdcall**, **__fastcall** и т.п.). Они влияют на способ управления параметрами при вызове функции (в каком порядке они передаются, кто их снимает со стека – вызывающий метод или вызываемый, и т.п.), на преобразование имени функции после компиляции и т.д.

В других языках есть схожие по назначению модификаторы. Таким образом, если мы импортируем функцию из библиотеки, написанной на одном из этих языков, использование неправильного параметра приведет к краху при вызове импортируемой функции. Для функции MessageBox видим, что при описании использовался модификатор WINAPI (в принципе, для ОС Windows он соответствует **__stdcall**).

• **string** EntryPoint – позволяет задать имя импортируемой функции, если оно не совпадает с именем, описанным в коде C#.

• **bool** PreserveSig – управляет типом значения, возвращаемого импортируемой функцией (по умолчанию **true**).

Многие функции Windows API возвращают в результате своей работы значение HRESULT, которое либо сигнализирует об успешном выполнении функции (значение 0), либо об ошибке (ненулевой код ошибки Windows). Программист должен сам обрабатывать ошибки, и если он забыл это сделать, то дальнейшее функционирование программы может стать нестабильным. Если данный флаг установить в значение **false**, то возвращаемым значением импортируемой функции станет **void**, а если при вызове функция вернет ненулевое значение, среда CLR сгенерирует исключение с сообщением об ошибке.

Пример:

```
      [DllImport("user32.dll", CharSet = CharSet.Unicode, EntryPoint =
"MessageBoxA")] static extern int MessageBoxA1(int hwnd, string msg, string
caption, uint flags);
      [DllImport("user32.dll", CharSet = CharSet.Ansi, EntryPoint =
"MessageBoxA")] static extern int MessageBoxA2(int hwnd, string msg, string
caption, uint flags);
      [DllImport("user32.dll", CharSet = CharSet.Unicode, EntryPoint =
"MessageBoxW")] static extern int MessageBoxW1(int hwnd, string msg, string
caption, uint flags);
      [DllImport("user32.dll", CharSet = CharSet.Ansi, EntryPoint =
"MessageBoxW")] static extern int MessageBoxW2(int hwnd, string msg, string
caption, uint flags);
```

```
static int Main()
{
      MessageBoxA1(0, "Привет от MessageBoxA №1!", "Сообщение", 0);
      MessageBoxA2(0, "Привет от MessageBoxA №2!", "Сообщение", 0);
      MessageBoxW1(0, "Привет от MessageBoxW №1!", "Сообщение", 0);
      MessageBoxW2(0, "Привет от MessageBoxW №2!", "Сообщение", 0);
      return 0;
}
```

Первое и четвертое сообщения будут нечитаемыми, т.к. вызываемым функциям будет передана строка в неподходящей кодировке. В принципе, значения именованных параметров атрибута DllImport по умолчанию настроены так, что почти всегда подходят для вызова функций API Windows. Поэтому можно просто написать

```
[DllImport("user32.dll")] static extern int MessageBox(int hwnd, string
msg, string caption, uint flags);
```

и в дальнейшем без проблем использовать данный метод MessageBox. Для нестандартных библиотек может потребоваться изменение некоторых параметров.

5.1.1.2. Импорт констант

Четвертый параметр функции MessageBox определяет дополнительные флаги – набор кнопок окна сообщения, картинка, кнопка по умолчанию, модальность и т.д. Возвращаемое значение – идентификатор нажатой кнопки. Они определены в том же заголовочном файле winuser.h. Вот часть из них:

```
#define IDOK                1
#define IDCANCEL            2
#define IDABORT             3
#define IDRETRY             4
#define IDIGNORE            5
#define IDYES               6
#define IDNO                7
#define IDCLOSE             8

#define MB_OK                       0x00000000L
#define MB_OKCANCEL                 0x00000001L
#define MB_ABORTRETRYIGNORE         0x00000002L
#define MB_YESNOCANCEL              0x00000003L
#define MB_YESNO                    0x00000004L
#define MB_RETRYCANCEL              0x00000005L

#define MB_ICONHAND                 0x00000010L
#define MB_ICONQUESTION             0x00000020L
#define MB_ICONEXCLAMATION          0x00000030L
#define MB_ICONASTERISK             0x00000040L
```

Импортировать их из библиотеки напрямую нельзя, поэтому можно

объявить в нашей программе именованные константы, присвоив им требуемые значения. Можно использовать обычные константы, но именованными пользоваться удобнее. Например:

```
      const int IDOK = 0x0001;
      const int IDCANCEL = 0x0002;
      const uint MB_OKCANCEL = 0x00000001;
      const uint MB_ICONQUESTION = 0x00000020;

      static int Main()
      {
            int rez = MessageBox(0, "Привет от MessageBox!", "Сообщение",
MB_ICONQUESTION | MB_OKCANCEL);
            if (rez == IDOK) Console.WriteLine("Вы нажали клавишу OK");
            else Console.WriteLine("Вы нажали клавишу Отмена");
            return 0;
      }
```

Увидим окно сообщения с картинкой вопроса и кнопками «ОК» и «Отмена».

5.5.1.3. Использование функций обратного вызова

Не только функции DLL могут быть вызваны из кода на языке C#, но и сами функции DLL могут вызывать определенные методы из нашего приложения в сценариях с обратными вызовами (callback). Сценарии обратного вызова включают в себя использование набора функций Win32 EnumXXX. При вызове этих функции для перечисления элементов им передается указатель на функцию, которая будет вызываться Windows каждый раз, когда искомый элемент будет найден. Это делается комбинированием PInvoke (для вызова функции DLL) и делегатов (для определения обратного вызова).

Например, данный код выполняет перечисление всех окон в системе и вывод их заголовков и имен классов:

```
      [DllImport("user32.dll")] static extern int GetWindowText(int hwnd,
StringBuilder text, int count);
      [DllImport("user32.dll")] static extern int GetClassName(int hwnd,
StringBuilder text, int count);
      [DllImport("user32.dll")] static extern int EnumWindows(EnumWindowProc
callback, int param);
      delegate bool EnumWindowProc(int hwnd, int param);

      static bool OutWindowCaption(int hwnd, int param)
      {
            StringBuilder text = new StringBuilder(1024);
            GetWindowText(hwnd, text, 1024);
            if (text.Length > 0)
            {
                  Console.Write("  {0}", text);
                  GetClassName(hwnd, text, 1024);
```

```
                Console.WriteLine(" ({0})", text);
            }
            return true;
        }

        static int Main()
        {
            EnumWindowProc proc = OutWindowCaption;
            Console.WriteLine("Активные окна в системе:");
            EnumWindows(proc, 0);
            return 0;
        }
```

Справку по этим функциям и назначению их параметров можно найти в библиотеке MSDN. Отметим лишь, почему для передачи параметра text используется класс StringBuilder. Значение этому параметру присваивается в вызываемой функции. Если бы это был числовой параметр, то мы передали бы его по ссылке или указателю. А класс **string** модификации строки не допускает. Поэтому резервируем 1024 символа для длины строки (можно узнать точное значение длины заголовка окна, но не будем усложнять программу) и используем класс для модифицируемых строк – Text.StringBuilder.

5.5.2. Написание небезопасного кода

Опасность и ненадежность небезопасного кода вовсе не являются его врожденными чертами. *Небезопасный код* – это код, выделение и освобождение памяти для которого не контролируется исполняющей средой .NET. Небезопасный код дает особенно заметные преимущества при использовании указателей для взаимодействия с написанным ранее кодом (таким как API Windows для языка C), или когда нашему приложению требуется прямое манипулирование памятью (как правило, из соображений повышения производительности).

Мы можете писать небезопасный код, применяя ключевые слова **unsafe**, **fixed** и **stackalloc**. Первое указывает, что помеченный им класс, член класса или блок будет работать в неуправляемом контексте. Второе отвечает за фиксацию управляемых объектов. По мере исполнения приложения выделенная под объекты память освобождается, в итоге остаются «фрагменты» свободной памяти. Чтобы не допустить фрагментации памяти, исполняющая среда .NET перемещает объекты, обеспечивая максимально эффективное использование памяти. Фиксация налагает запрет на перемещение данного объекта сборщиком мусора (garbage collector, GC). Третье указывает, что дина-

мический блок памяти будет выделяться не в куче (как это делает оператор **new**), а на стеке.

 Пример: Samples\5.5\5_5_2_unsafe.

5.5.2.1. Использование модификатора unsafe

Рассмотрим некоторые правила, которые относятся к использованию указателей. В принципе, многие правила заимствованы из языка C++. Использование указателей допускается только в небезопасном коде. Т.е. модификатором **unsafe** могут быть помечены:

• Блок кода (т.е. здесь **unsafe** выступает как внедряемый оператор, а не модификатор). В этом случае внутри этого блока разрешено использование указателей и объявление локальных переменных, имеющих тип указателя. Синтаксис:

```
unsafe <блок>
```

• Поле класса (а также свойство, индексатор и т.п.). В этом случае типом этого поля (свойства, индексатора) может быть указатель.

• Метод (в т.ч. конструктор, деструктор). В этом случае в данном методе можно работать с указателями и объявлять локальные переменные типа указателя, а также его параметры и возвращаемое значение могут иметь тип указателя.

• Тип (класс, структура или интерфейс). В этом случае считается, что все члены типа объявлены с модификатором **unsafe**.

Синтаксис объявления указателей следующий:

```
<объявление указателей> :: <тип данных>*[<тип массива>] <список
переменных с инициализацией>;
```

Здесь тип данных – это любой тип данных по значению, тип указателя или **void**. Указатели на структуры допускаются в том случае, если в структуре (а также во вложенных в нее структурах, если таковые имеются) не содержатся поля ссылочного типа. Пример:

```
struct S
{
      public int a;
      public double b;
}

struct T
{
```

```
        public int a;
        public double b;
        public string s;
    }

    unsafe int* iptr; // Указатель на int
    unsafe string* str; // Ошибка
    unsafe int** dptr; // Указатель на указатель на int
    unsafe char*[,] pch; // Массив указателей на char
    unsafe void* vp; // Указатель на любой тип данных
    unsafe S* sptr; // Указатель на S
    unsafe T* tptr; // Ошибка
```

Заметьте, что в отличие от языка C++, при объявлении указателя символ «*» считается частью типа. Например, в языке C++ объявление двух указателей может выглядеть так:

```
    int *a = NULL, *b = NULL;
```

Для обозначения нулевого указателя в языке C# используется, как и для пустых ссылок, значение **null**, поэтому аналогичное объявление будет выглядеть следующим образом:

```
    int* a = null, b = null;
```

Не забудьте разрешить использование небезопасного кода в проекте (пункт меню «Проект» → «Свойства…», вкладка «Построение», опция «Разрешить небезопасный код»).

5.5.2.2. Выражения с указателями

Операции, используемые в выражениях для аргументов типа указателей, перечислены в табл. 5.5.

Табл. 5.5 – Операторы для работы с указателями в языке C#

Оператор	Описание
&	Оператор получения адреса (address-of). Возвращает указатель на экземпляр типа. Аргумент должен быть l-value. Допускается получать указатели на типы по значению (для структур действует упомянутое выше ограничение), а также массивы. Для массивов, строго говоря, мы получаем указатель на первый элемент массива, поэтому он должен иметь допустимый тип.
*	Оператор разыменования (dereference) обеспечивает доступ к значению, на которое ссылается указатель. Аргументом опе-

	рации должен являться указатель любого типа, кроме **void***.
`->`	Оператор разыменования и доступа к члену, если имеем указатель на структуру. Т.е. запись «`x->y`» соответствует записи «`(*x).y`».
`[]`	Оператор разыменования и доступа к элементу. Запись «`x[i]`» соответствует записи «`*(x + i)`». Применим к указателям любого типа, кроме **void***. Параметр «i» должен иметь тип **int**, **uint**, **long** или **ulong**.
`()`	Явное преобразование типа. Возможно неявное преобразование любого указателя к типу **void***, и значения **null** к любому указателю. Явно можно преобразовать указатель одного типа к указателю другого типа, а также целый тип к указателю или указатель к целому типу.
`++ --`	Для указателя типа T* увеличивает или уменьшает значение адреса на **sizeof**(T). Не применяется к указателям типа **void***.
`+ -`	Сложение указателя типа T* со значением «i», имеющим тип **int**, **uint**, **long** или **ulong**, увеличит значение адреса на i***sizeof**(T). Вычитание такого значения из указателя, соответственно, уменьшит значение адреса на эту величину. Разность двух указателей даст величину типа **long**, представляющую собой разность адресов между этими указателями, деленную на **sizeof**(T).
`< <= > >= == !=`	Сравнение указателей.
`sizeof`	Возвращает количество байтов, отводимых в памяти для хранения переменной типа указателя. Разрядность указателя связана с разрядностью ОС, поэтому размер указателя для архитектуры x86 или x64 равен **sizeof**(**int**).

Пример:

```
unsafe static void Swap1(int* a, int* b)
{
      int c = *a;
      *a = *b;
      *b = c;
}

unsafe static void Swap2(int* a, int* b)
{
```

```
            int c = a[0];
            a[0] = b[0];
            b[0] = c;
      }

      unsafe static void FillS1(S* s)
      {
            s->a = 5;
            s->b = 7.77;
      }

      unsafe static void FillS2(S* s)
      {
            (*s).a = 7;
            (*s).b = 5.55;
      }

      static int Main()
      {
            int x = 5, y = 7;

            unsafe
            {
                  Swap1(&x, &y);
                  Console.WriteLine("x = {0}, y = {1}", x, y); // x = 7, y = 5
                  Swap2(&x, &y);
                  Console.WriteLine("x = {0}, y = {1}", x, y); // x = 5, y = 7
            }

            return 0;
      }
```

Функции Swap1 и Swap2 обменивают между собой значения аргументов. При этом во втором случае используется тот факт, что «`x[0] = *(x + 0) = *x`». Процедуры FillS1 и FillS2 демонстрируют разные способы обращения к полям структуры через указатель.

5.5.2.3. Оператор fixed

Как было сказано выше, этот оператор говорит сборщику мусора о том, чтобы тот не перемещал заданные переменные в памяти. Некоторые переменные являются фиксированными по умолчанию:

1. Локальные переменные. Они помещаются на стек, и их адрес не изменяется. Поэтому в предыдущем примере компилятор разрешил получение адреса локальных переменных. Когда время жизни локальной переменной заканчивается, она удаляется со стека.

2. Поле структуры (`S.F`), если структура фиксирована в памяти (т.е. объявлена как локальная).

3. Переменная, полученная в результате разыменования указателя (`*x`

или `x[i]`) или доступа к полю структуры по указателю (`S->F`). Это тоже видно по предыдущему примеру.

С указателем можно работать только в том случае, если он фиксирован в памяти. Иначе нет гарантии, что при обращении к указателю данные будут расположены по требуемому адресу. При попытке использования нефиксированных указателей компилятор C# генерирует ошибку. Например:

```
unsafe static void FillS3(ref S s)
{
      (&s)->a = 7; // Ошибка
      (&s)->b = 5.55; // Ошибка

      fixed (S* p = &s)
      {
            p->a = 7; // OK
            p->b = 5.55; // OK
      }
}
```

Синтаксис внедряемого оператора **fixed**:

```
fixed (<объявление указателей>) <блок>
```

Внутри блока объявленные указатели будут фиксированными. Пример:

```
static void Proc(ref int a, ref int b)
{
      unsafe
      {
            fixed (int* start = &mas[0], pa = &a, pb = &b)
            {
                  Console.WriteLine("Переданы указатели на элементы
массива №{0} и №{1}", pa - start, pb - start);
                  Console.WriteLine("Элементы, расположенные между
ними:");

                  for (int* p = pa; (pa <= pb && p <= pb) || (pa > pb &&
p >= pb); p += pa <= pb ? 1 : -1)
                  {
                        Console.Write("{0} ", *p);
                  }

                  Console.WriteLine();
            }
      }
}

static S s1;
static int[] mas = { 1, 2, 3, 4 };

static int Main()
{
      S s2;

      unsafe
      {
```

```
                int* p1 = &s1.a; // Ошибка
                int* p2 = &s2.a; // ОК
                Proc(ref mas[3], ref mas[1]);
            }

            return 0;
        }
```

Вывод на консоль:

```
Переданы указатели на элементы массива №3 и №1
Элементы, расположенные между ними:
4 3 2
```

Также ключевое слово **fixed** используется в качестве модификатора полей структур в небезопасном контексте для объявления буферов фиксированного размера. Это приблизительно соответствует объявлению одномерных массивов фиксированного размера в языке C++:

```
    [<атрибуты>] [<модификаторы>] fixed <тип> {<идентификатор>
"["<константа>"]"} [, ...];
```

В качестве типа элементов буфера допускаются все типы по значению, кроме **decimal**, структур и перечислений. Допустимые модификаторы – модификаторы доступа, а также **new** и **unsafe**. Пример:

```
    unsafe struct Fixed
    {
        fixed char Buf[50];

        public Fixed(string str)
        {
            fixed (char* ch = &str.ToCharArray()[0], buf = Buf)
            {
                int i;
                for (i = 0; i < str.Length && i < 49; i++)
                {
                    buf[i] = ch[i];
                }
                buf[i] = '\0';
            }
        }

        public void Out()
        {
            fixed (char *b = Buf)
            {
                Console.Write("Символы в буфере:");
                for (int i = 0; b[i] != 0; i++) Console.Write(" {0}",
b[i]);
                Console.WriteLine();
            }
        }
    }

    static int Main()
```

```
{
      Fixed f = new Fixed("Привет!");
      f.Out();
      return 0;
}
```

Результаты работы программы:

```
Символы в буфере: П р и в е т !
```

5.5.2.4. Оператор stackalloc

Оператор выражения **stackalloc** используется в небезопасном контексте кода для выделения фиксированного блока памяти в стеке. Синтаксис:

```
stackalloc <тип> "["<выражение>"]"
```

Выражение должно иметь тип **int**. Результатом работы данного оператора будет указатель на блок памяти размером **sizeof**(<тип>)*<выражение> байт, расположенный на стеке. Содержимое вновь выделенной памяти неопределенное. Память автоматически освобождается при завершении блока, в котором была выделена память. Пример:

```
unsafe static void Proc()
{
      int* p = stackalloc int[10];

      p[0] = 1;
      p[1] = 1;
      for (int i = 2; i < 10; i++) p[i] = p[i - 1] + p[i - 2];
      Console.Write("Числа Фибоначчи:");
      for (int i = 0; i < 10; i++) Console.Write(" {0}", *(p + i));
      Console.WriteLine();
}

static int Main()
{
      Proc();
      return 0;
}
```

Здесь вместо выражения «`p[i]`» показано использование аналогичного ему выражения «`*(p + i)`». Вывод на консоль:

```
Числа Фибоначчи: 1 1 2 3 5 8 13 21 34 55
```

Для динамического выделения памяти в куче можно использовать функции Windows API, такие, как HeapAlloc, HeapFree и т.п., импортировав их из библиотеки kernel32.dll.

§ 5.6. Комментарии и документирование кода

В этом разделе мы опишем приемы комментирования исходного кода. В C# используется три типа комментирования: комментирование блока текста, комментирование строки и XML-комментирование.

5.6.1. Комментирование кода

Для комментирования строки текста используется комбинация символов «//». В этом случае все символы, от данной комбинации и до конца строки, становятся частью комментария. Например:

```
// Убираем предупреждения компилятора  о том, что некоторые переменные
// ни разу не использованы
#pragma warning disable 0168
```

или

```
int i; // это счетчик цикла
```

Для комментирования блока текста используются комбинации символов «/*» и «*/». В этом случае все символы, заключенные между ними, считаются частью комментария:

```
/*
      Программа для демонстрации комментариев
*/
```

При этом комментарий, описанный внутри другого комментария, рассматривается просто как набор символов. Например, в данном случае

```
int i; // это счетчик цикла  /*
int k; */
```

комбинации символов «/*» и «*/» не описывают комментарий, т.к. первая из них попадает в область действия другого комментария. Поэтому компилятор покажет ошибку – «Недопустимый элемент "/" в выражении». Другой пример с ошибкой:

```
/*int i; // это счетчик*/ цикла
```

Здесь, наоборот, в область действия блока комментария попадает строковый комментарий, поэтому символы «//» рассматриваются просто как символы. И область действия комментария не будет распространена до конца строки, а закончится комбинацией символов «*/».

Следующий синтаксис допустим:

```
/*
int k; // а это комментарий внутри другого комментария
*/
int x; // /* еще один вложенный комментарий */
```

В этих случаях, если символы внешнего комментария будут удалены, вложенные комментарии станут самостоятельными комментариями:

```
int k; // а это комментарий внутри другого комментария
int x; /* еще один вложенный комментарий */
```

Этот прием часто используется. Если небольшая часть кода в программе временно не нужна (часть строки или строка), ее заключают в строковый комментарий. Если необходимо этим способом закомментировать несколько строк, их можно выделить и использовать специальную кнопку на панели инструментов «Закомментировать выделенные строки» (Ctrl+E+C). Нажатие кнопки «Отменить преобразование выделенных строк в комментарий» (Ctrl+E+U) выполнит обратное действие. Соответствующие пункты меню находятся в меню «Правка» → «Дополнительно». Затем, если необходимо закомментировать более крупный кусок кода, содержащий в себе строковые комментарии, используется комментарий блока. Например:

```
/*int j; // еще один счетчик
int sum = 0; // здесь будем накапливать сумму

for (j = 1; j < i; j++)
{
      sum += j;
}*/
```

В случае необходимости внешний комментарий быстро убирается, и код снова готов для отладки. Однако, что делать, если нам необходимо закомментировать большой блок кода, уже содержащий в себе оба типа комментариев? Заключить его в еще один блочный комментарий не получится:

```
/*for (i = 0; i <= 5; i++)
{
      Console.WriteLine(i);
}

/*for (i = 5; i >= 0; i--)
{
      Console.WriteLine(i);
}*/

int j; // еще один счетчик
int sum = 0; // здесь будем накапливать сумму

for (j = 1; j < i; j++)
{
      sum += j;
```

```
    }

    Console.WriteLine("Сумма = {0}", sum);*/
```

Итак, мы пытались весь приведенный код заключить во внешний блочный комментарий. Это не удалось, т.к. в коде содержался другой блочный комментарий, и его завершающая комбинация символов «*/» завершила область действия внешнего комментария. В этом случае можно использовать директиву условной компиляции **#if/#endif**. Так как весь код, попадающий в область действия этой директивы, когда ее условие не выполняется, исключается из компиляции, достаточно указанную область кода поместить между директивами

```
#if false
      // код, который нужно исключить из компиляции
#endif
```

 Пример: Samples\5.6\5_6_1_comment.

5.6.2. XML-документирование кода C#

 Пример: Samples\5.6\5_6_2_xml.

5.6.2.1. Получение XML-кода

Третьим способом комментирования является XML-документирование кода C#. Для того, чтобы при компиляции генерировался XML-файл документации, необходимо использовать опцию компилятора «/doc»:

```
csc.exe /doc:<имя файла> ...
```

Или, при компиляции из среды разработки, зайти в свойства проекта («Проект» → «Свойства»), выбрать закладку «Построение» и отметить опцию «XML-файл документации». В расположенном рядом поле можно ввести имя результирующего файла XML.

Комментарий к коду XML, аналогично обычным комментариям, может быть строчным или блочным. Строчный комментарий распространяет свое действие от комбинации символов «///» до конца строки. Блочный комментарий – от комбинации символов «/**» до «*/». Помимо этого, компилятор проверяет синтаксис некоторых тегов XML и допустимость употребления большинства тегов. Например, если мы опишем несуществующий аргумент метода, компилятор выдаст ошибку.

В получаемом XML-файле все элементы кода имеют некоторые правила представления. Во-первых, при упоминании элемента кода используется полное иерархическое представление пространств имен. Например, если мы использовали тип String, то его представление в XML будет иметь вид System.String, точнее – T:System.String, т.к. перед каждым элементом кода в XML ставится модификатор типа элемента (табл. 5.6).

Табл. 5.6 – Модификаторы элементов XML-кода

Модификатор	Описание
N	Пространство имен
T	Тип (класс, интерфейс, структура, перечисление, делегат)
F	Поле
P	Свойство или индексатор
M	Метод (далее под методами подразумеваются также конструкторы, деструкторы и операторы)
E	Событие
!	Строка ошибки

Допустимые теги перечислены в табл. 5.7.

Табл. 5.7 – Теги XML-документации

Тег	Описание
`<c>`	Текст является кодом
`<code>`	То же самое, но для маркировки нескольких строк
`<example>`	Приводится пример использования кода
`<exception cref="класс">`(*)	Указание возможных исключений в типах и методах. Параметр `cref` должен содержать ссылку на класс исключения (Exception или его потомок)
`<include>`(*)	Используется для ссылок на комментарии в другом файле
`<list type="тип">`	Используется для создания таблицы (тип `table`) или списка (`bullet` или `number`). Внутри тега `<list>` можно использовать тег `<listhead>` для описания заголовка и тег `<item>` для элемента таблицы или списка. Внутри данных тегов используются теги `<term>` и

	<description> для описания терминов и определений
<para>	Используется для разбивки текста на параграфы
<param name="имя">(*)	Используется для описания аргументов методов
<paramref>	Ссылка на параметр метода
<permission cref="класс">(*)	Комментирование прав доступа к типу или члену. Параметр cref должен содержать ссылку на класс разрешений доступа (PermissionSet или его потомок)
<remarks>	Дополнительные сведения, отображающиеся в обозревателе объектов
<returns>	Описание возвращаемого методом значения
<see cref="имя"/>(*)	Ссылка на другой тип или член документации
<seealso cref="имя"/>(*)	Ссылки для раздела «См. также»
<summary>	Описание типа или члена типа
<typeparam>(*)	Описание параметра универсального типа или метода
<typeparamref>	Ссылка на параметр универсального типа или метода
<value>	Описание свойства или индексатора

Примечания: (*) – синтаксис тега проверяется компилятором.

Угловые скобки считаются частью кода XML, поэтому если их необходимо использовать в качестве текста или значения параметра, используются мнемокоды «<» и «>». Можно использовать и некоторые другие мнемокоды языка HTML. Для вставки символа с указанным кодом используется мнемокод «&#<код>;». Например, мнемокод для неразрывного пробела – « ».

Пример:

```
namespace XMLDocumentationSample
{
      /// <summary>
      /// Главный класс программы <c>Program</c>.
      /// </summary>
      /// <permission cref="System.Security.PermissionSet">
      /// Доступ к классу не ограничен.</permission>
      public class Program
      {
            /// <summary>
            /// <para>Метод <c>Mul2</c> возвращает результат умножения
            /// аргументов <paramref name="a"/> и
            /// <paramref name="b"/>.</para>
            /// <para>Существует также версия для трех аргументов
```

```
/// (<see cref="Mul3"/>).</para>
/// </summary>
/// <param name="a">Первый множитель.</param>
/// <param name="b">Второй множитель.</param>
/// <returns>Результат умножения
/// <paramref name="a"/>*<paramref name="b"/>.</returns>
/// <exception cref="OverflowException">Если результат
/// умножения <paramref name="a"/> и <paramref name="b"/>
/// приводит к переполнению.</exception>
/// <example><para>Пример использования:</para>
/// <code>
/// int a = 5, b = 5, c;
/// try
/// {
///     c = Program.Mul2(a, b);
/// }
/// catch(OverflowException)
/// {
///     Console.WriteLine("Результат
/// умножения приводит к переполнению");
/// }
/// </code>
/// </example>
/// <permission cref="System.Security.PermissionSet">
/// Доступ к методу не ограничен.</permission>
/// <seealso cref="System.Exception"/>
/// <seealso cref="System.Int32"/>
public static int Mul2(int a, int b)
{
      return checked(a * b);
}

/// <summary>
/// <para>Метод <c>Mul3</c> возвращает результат умножения
/// аргументов <paramref name="a"/>, <paramref name="b"/>
/// и <paramref name="c"/>.</para>
/// <para>Существует также версия для двух аргументов
/// (<see cref="Mul2"/>).</para>
/// </summary>
/// <param name="a">Первый множитель.</param>
/// <param name="b">Второй множитель.</param>
/// <param name="c">Третий множитель.</param>
/// <returns>Результат умножения <paramref name="a"/>*
/// <paramref name="b"/>*<paramref name="c"/>.</returns>
/// <exception cref="OverflowException">Если результат
/// умножения <paramref name="a"/>, <paramref name="b"/>
/// и <paramref name="c"/> приводит к переполнению.
/// </exception>
/// <example><para>Пример использования:</para>
/// <code>
/// int a = 5, b = 5, c = 5, d;
/// try
/// {
///     d = Program.Mul3(a, b, c);
/// }
/// catch(OverflowException)
/// {
///     Console.WriteLine("Результат
/// умножения приводит к переполнению");
/// }
```

```
        /// </code>
        /// </example>
        /// <permission cref="System.Security.PermissionSet">
        /// Доступ к методу не ограничен.</permission>
        /// <seealso cref="System.Exception"/>
        /// <seealso cref="System.Int32"/>
        public static int Mul3(int a, int b, int c)
        {
            return checked(a * b * c);
        }

        /// <summary>
        /// Точка входа приложения.
        /// </summary>
        /// <param name="args">Аргументы командной строки.</param>
        /// <returns>Код возврата приложения:
        /// <list type="table">
        ///   <listheader>
        ///     <term>Код</term>
        ///     <term>Описание</term>
        ///   </listheader>
        ///   <item>
        ///     <term>0</term>
        ///     <term>Успешное завершение приложения</term>
        ///   </item>
        ///   <item>
        ///     <term>1</term>
        ///     <term>Приложение завершено с ошибкой</term>
        ///   </item>
        /// </list>
        /// </returns>
        static int Main(string[] args)
        {
            try
            {
                Console.WriteLine(Mul2(5, 5));
                Console.WriteLine(Mul3(5, 5, 5));
            }
            catch
            {
                return 1;
            }

            return 0;
        }
    }
}
```

В результате работы программы генерируется следующий код XML:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>5_6_2_xml</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentationSample.Program">
            <summary>
            Главный класс программы <c>Program</c>.
            </summary>
```

```
                <permission cref="T:System.Security.PermissionSet">Доступ к
классу не ограничен.</permission>
            </member>
            <member
name="M:XMLDocumentationSample.Program.Mul2(System.Int32,System.Int32)">
                <summary>
                <para>Метод <c>Mul2</c> возвращает результат умножения
аргументов
                <paramref name="a"/> и <paramref name="b"/>.</para>
                <para>Существует также версия для трех аргументов (<see
cref="M:XMLDocumentationSample.Program.Mul3(System.Int32,System.Int32,System.
Int32)"/>).</para>
                </summary>
                <param name="a">Первый множитель.</param>
                <param name="b">Второй множитель.</param>
                <returns>Результат умножения <paramref name="a"/>*<paramref
name="b"/>.</returns>
                <exception cref="T:System.OverflowException">Если результат
умножения <paramref name="a"/> и
                <paramref name="b"/> приводит к переполнению.</exception>
                <example><para>Пример использования:</para>
                <code>
                int a = 5, b = 5, c;
                try
                {
                    c = Program.Mul2(a, b);
                }
                catch(OverflowException)
                {
                    Console.WriteLine("Результат умножения приводит к
переполнению");
                }
                </code>
                </example>
                <permission cref="T:System.Security.PermissionSet">Доступ к
методу не ограничен.</permission>
                <seealso cref="T:System.Exception"/>
                <seealso cref="T:System.Int32"/>
            </member>
            <member
name="M:XMLDocumentationSample.Program.Mul3(System.Int32,System.Int32,System.
Int32)">
                <summary>
                <para>Метод <c>Mul3</c> возвращает результат умножения
аргументов
                <paramref name="a"/>, <paramref name="b"/> и <paramref
name="c"/>.</para>
                <para>Существует также версия для двух аргументов (<see
cref="M:XMLDocumentationSample.Program.Mul2(System.Int32,System.Int32)"/>).</
para>
                </summary>
                <param name="a">Первый множитель.</param>
                <param name="b">Второй множитель.</param>
                <param name="c">Третий множитель.</param>
                <returns>Результат умножения <paramref name="a"/>*<paramref
name="b"/>*<paramref name="c"/>.</returns>
                <exception cref="T:System.OverflowException">Если результат
умножения <paramref name="a"/>,
                <paramref name="b"/> и <paramref name="c"/> приводит к
переполнению.</exception>
```

```
                    <example><para>Пример использования:</para>
                    <code>
                    int a = 5, b = 5, c = 5, d;
                    try
                    {
                        d = Program.Mul3(a, b, c);
                    }
                    catch(OverflowException)
                    {
                        Console.WriteLine("Результат умножения приводит к
переполнению");
                    }
                    </code>
                    </example>
                    <permission cref="T:System.Security.PermissionSet">Доступ к
методу не ограничен.</permission>
                    <seealso cref="T:System.Exception"/>
                    <seealso cref="T:System.Int32"/>
                </member>
                <member
name="M:XMLDocumentationSample.Program.Main(System.String[])">
                    <summary>
                    Точка входа приложения.
                    </summary>
                    <param name="args">Аргументы командной строки.</param>
                    <returns>Код возврата приложения:
                    <list type="table">
                      <listheader>
                        <term>Код</term>
                        <term>Описание</term>
                      </listheader>
                      <item>
                        <term>0</term>
                        <term>Успешное завершение приложения</term>
                      </item>
                      <item>
                        <term>1</term>
                        <term>Приложение завершено с ошибкой</term>
                      </item>
                    </list>
                    </returns>
                </member>
            </members>
        </doc>
```

5.6.2.2. Генерация документации

Далее можно полученный код XML преобразовать в документацию любого формата. В ранних версиях Visual Studio был встроенный инструмент для преобразования XML в HTML. В настоящее время можно использовать внешние инструменты. Например, утилиту NDoc (сайт ndoc.sourceforge.net). К учебнику прилагается дистрибутив этой утилиты версии 1.3.1 (размер дистрибутива – около 2,7 Мб). К сожалению, ее развитие остановилось на версии .NET 1.1. Если проект использует более старшую версию .NET, восполь-

зоваться этой утилитой не удастся.

До появления .NET, корпорация Microsoft для создания файлов справки распространяла утилиту HTML Help Workshop. Полученные с ее помощью файлы формата CHM (Microsoft Compiled/Compressed HTML) содержат набор HTML-страниц с перекрестными ссылками и дополнительные ресурсы (такие, как картинки). Также существует возможность создания оглавления, организации поиска по индексу и т.д. Также существует HTML Help API для интеграции файла справки CHM с приложением (открытие справки на нужной странице, привязка визуальных элементов управления программы к контексту справки и т.п.).

В настоящее время для получения файлов CHM в среде .NET используется утилита Microsoft Sandcastle. Для ее использования необходимо установить собственно генератор документации Sandcastle (свежая версия доступна на сайте sandcastle.codeplex.com, к учебнику прилагается дистрибутив версии 2.6 за июнь 2010 года, 15 Мб) и визуальную среду разработки Sandcastle Help File Builder (shfb.codeplex.com, к учебнику прилагается дистрибутив версии 1.9.1.0, 9 Мб). В отличие от многих сторонних утилит, Sandcastle умеет работать с отражением, и извлекает из сборок .NET всю необходимую для документирования информацию. Кроме того, документация может быть сохранена не только в формате HTML Help, но и в формате MSHelp 2, а также в виде набора страниц HTML (статических и с динамическим поиском).

Итак, установим на ПК Sandcastle и SHFB. Создадим в папке проекта «5_6_2_xml» подпапку «html». Запустим утилиту SHFB, для удобства можно (подобно тому, как это было сделано в п. 2.3.6) добавить в меню «Сервис» среды разработки Visual Studio дополнительный пункт для запуска SHFB, исполняемый файл утилиты по умолчанию имеет спецификацию

```
      <папка Program Files>\EWSoftware\Sandcastle Help File
Builder\SandcastleBuilderGUI.exe
```

Создадим в SHFB новый проект (пункт меню «File» → «New Project», соответствующая кнопка на панели инструментов или комбинация клавиш Ctrl+N), сохраним его в ранее созданной папке «html» с любым именем. После этого в правой части окна появится панель «Project Explorer». В контекстном меню для пункта «Documentation Source» выберем единственный пункт «Add Documentation Source…» (рис. 5.3).

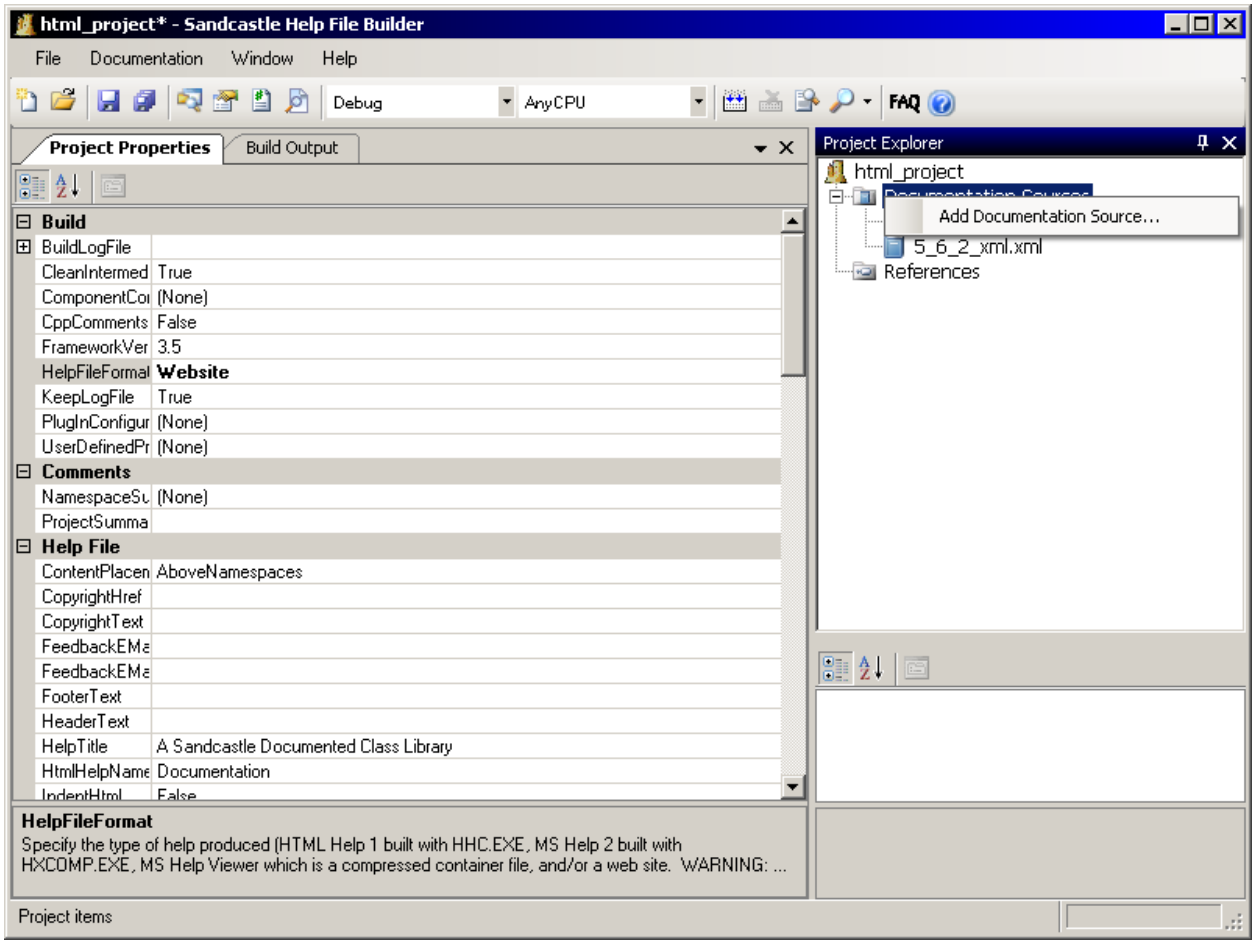


Рис. 5.3 – Создание документации в утилите SHFB

В появившемся диалоге открытия файла выберем исполняемый файл сборки «5_6_2_xml.exe». При этом автоматически сформируется XML-файл и также добавится в источники документации. После этого на странице «Project Properties» можно установить опции проекта. Выберем формат документации «HelpFileFormat = Website» (чтобы HTML-файлы не собирались в один CHM файл, а были доступны по отдельности) и язык «Language = Русский (Россия)». Все, документацию можно генерировать (пункт меню «Documentation» → «Build Project», соответствующая кнопка на панели инструментов или комбинация клавиш Ctrl+Shift+B). После этого в папке «html» появится подпапка «Help» со всеми файлами документации. Для просмотра выберите файл «Index.html» (рис. 5.4).

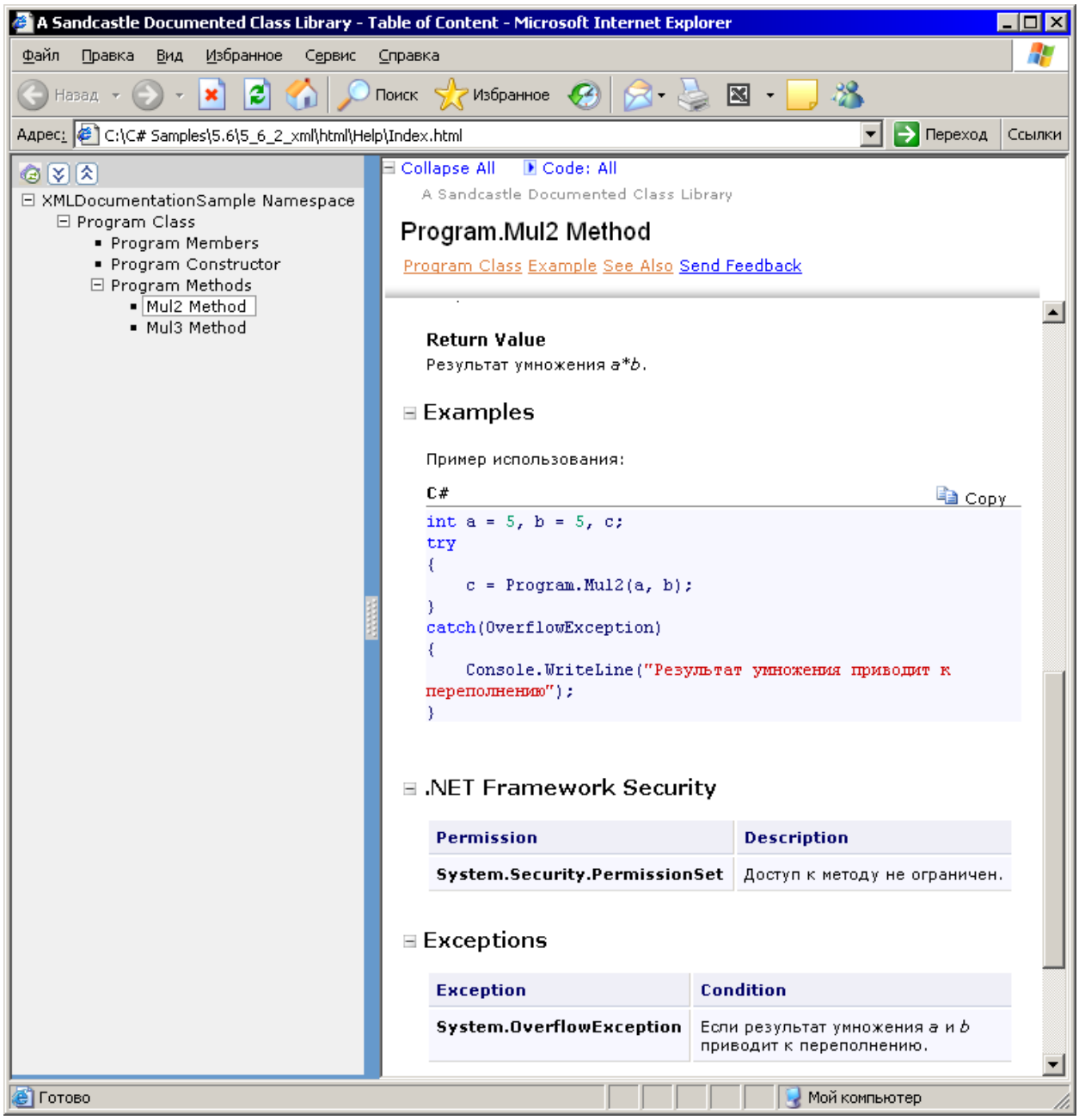


Рис. 5.4 – Готовая документация

5.6.2.3. Использование XML-кода в IntelliSense

Кроме того, XML-документацией может пользоваться среда разработки Visual Studio, используя технологию IntelliSense. При использовании документированных типов и членов информация из документации появляется во всплывающей подсказке (рис. 5.3).

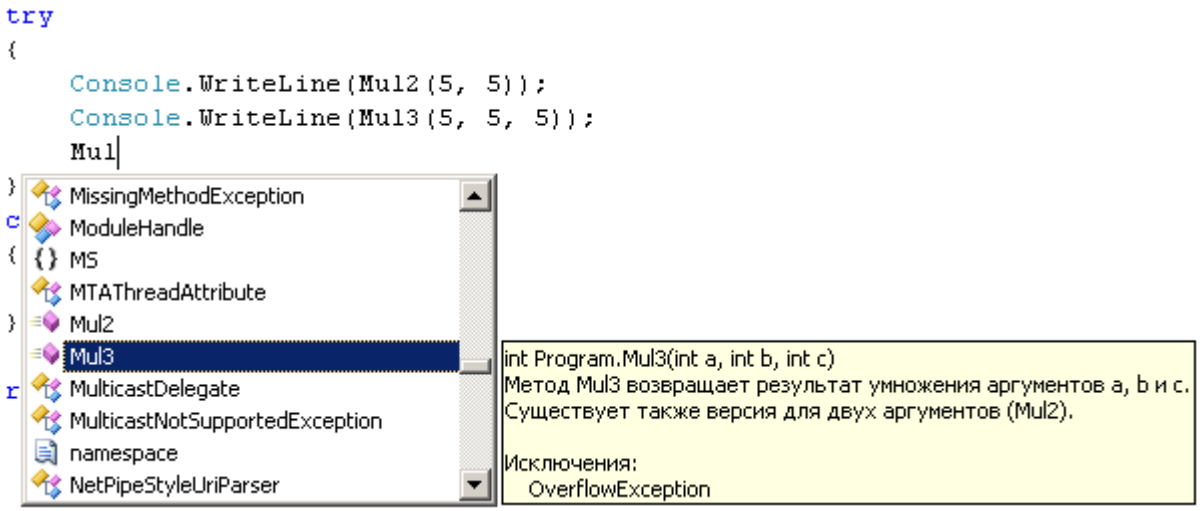


Рис. 5.5 – Использование документации по методу класса

Также появляется всплывающая подсказка по параметрам документированного метода (рис. 5.4).

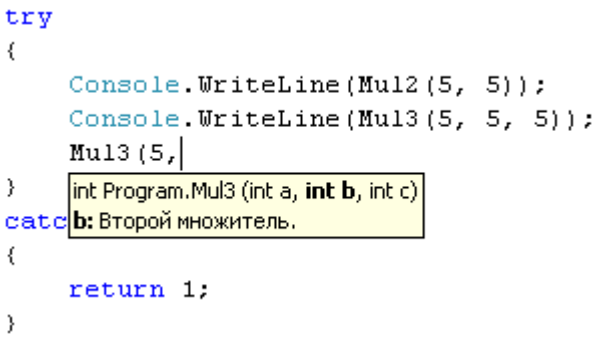


Рис. 5.6 – Использование документации по параметрам метода

6. ЗАКЛЮЧЕНИЕ

К сожалению, объем курса не позволяет рассмотреть:

• Некоторые полезные возможности среды .NET и языка C# (в частности, работу с регулярными выражениями, лямбда-выражения, выражения запросов LINQ – Language-Integrated Query);

• Технологии разработки сетевых приложений – Active Server Page (ASP.NET приложения), web-службы, программирование сокетов и т.п.;

• Управление базами данных (включая технологию доступа к данным ADO – Active Data Objects);

• Разработку оконных приложений (Window Forms, WPF) и технологию визуального программирования;

• Программирование графики (OpenGL, Direct3D, GDI+);

• Работу с интерфейсами COM (Component Object Model).

Сетевые технологии и технологии баз данных изучаются в рамках соответствующих дисциплин. Регулярные выражения, визуальное программирование и работа с мультимедиа изучаются в рамках дисциплины «Мультимедиа-технологии».

Количество классов .NET настолько велико, что описать их все в рамках одной, сколь угодно объемной, книги не представляется возможным. Самое главное, что мы рассмотрели каркас .NET Framework, принципы объектно-ориентированного программирования, базовые классы и один из языков .NET – C#, а также научились пользоваться справочной системой MSDN.

Все разработчики .NET стараются проектировать свои классы так, чтобы ими было удобно пользоваться – классы располагаются в одном пространстве имен со схожими по функциональности классами. Сами классы и их члены имеют интуитивно понятные имена, а также хорошо документированы. Поэтому освоение новых компонентов облегчено настолько, насколько это возможно.

Будем надеяться, что и разработанные вами компоненты будут удовлетворять этим критериям.

СПИСОК ЛИТЕРАТУРЫ

1. Библиотека Microsoft Developer Network (MSDN). – http://msdn.microsoft.com/ru-ru/library.

2. Биллиг В.А. Основы программирования на C#. – Интернет Университет Информационных технологий (ИНТУИТ). – http://www.intuit.ru/department/pl/csharp/.

3. Долженко А.И. Разработка Windows-приложений на языке C# 2005. – Интернет Университет Информационных технологий (ИНТУИТ). – http://www.intuit.ru/department/se/devcsharp2005/.

4. Кариев Ч.А. Создание Windows-приложений на основе Visual C#. – Интернет Университет Информационных технологий (ИНТУИТ). – http://www.intuit.ru/department/pl/visualcsharp/.

5. Марченко А.Л. Введение в программирование на C# 2.0. – Интернет Университет Информационных технологий (ИНТУИТ). – http://www.intuit.ru/department/pl/csharp20/.

6. Петцольд Ч. Программирование для Microsoft Windows на C#. – М.: Русская Редакция, 2002. – Том 1. – 576 с.

7. Петцольд Ч. Программирование для Microsoft Windows на C#. – М.: Русская Редакция, 2002. – Том 2. – 624 с.

8. Рихтер Дж. Программирование на платформе Microsoft .NET Framework. – М.: Русская Редакция, 2003. – 512 с.

9. Троелсен Э. C# и платформа .NET 3.0. – СПб: Питер, 2008. – 1456 с.

ПРИЛОЖЕНИЯ

Приложение А. Объекты для работы с датой и временем

Структура DateTime представляет текущее время, обычно выраженное как дата и время суток в диапазоне от 00:00:00 1 января 0001 года (н.э.) до 23:59:59 31 декабря 9999 года (н.э.).

Значения времени измеряются в 100-наносекундных единицах, называемых тактами, и точная дата представляется числом тактов с 00:00:00 1 января 0001 года н.э. Например, значение тактов, равное 31241376000000000L, представляет 00:00:00, пятницу 1 января 0100 года.

В таблице А.1 кратко рассмотрены основные открытые (**public**) члены структуры DateTime.

Табл. А.1 – Члены структуры DateTime

Член	Владелец	Описание
Конструкторы		
`DateTime(...)`		Имеется большой набор конструкторов, инициализирующих структуру различными наборами аргументов (см. справку MSDN)
Методы		
`static int` `Compare(DateTime t1, DateTime t2)`		Сравнивает два экземпляра объекта DateTime
`int` `CompareTo(`**`object`** `value)`	`IComparable`	Сравнивает значение даты и времени с другим объектом (см. п. 4.9.4)
`DateTime Add(TimeSpan value)`		Возвращает новый объект, добавляющий интервал времени TimeSpan(1) к значению данного экземпляра
`DateTime AddTicks(`**`long`** `value)`		Возвращает новый объект, добавляющий заданное число тактов к значению данного экземпляра

DateTime AddMilliseconds (**double** value)		То же для миллисекунд
DateTime AddSeconds (**double** value)		То же для секунд
DateTime AddMinutes (**double** value)		То же для минут
DateTime AddHours (**double** value)		То же для часов
DateTime AddDays (**double** value)		То же для дней
DateTime AddMonths (**int** months)		То же для месяцев
DateTime AddYears (**int** value)		То же для лет
static int DaysInMonth(**int** year, **int** month)		Возвращает число дней в указанном месяце указанного года
override bool Equals(**object** value)	Object	Возвращает значение, указывающее, равен ли данный экземпляр заданному объекту
static bool Equals(DateTime t1, DateTime t2)		Возвращает значение, указывающее, равны ли два экземпляра DateTime
Type GetType()	Object	Возвращает метакласс типа
TypeCode GetTypeCode()	IConvertible	Возвращает TypeCode для класса DateTime
static bool IsLeapYear(**int** year)		Возвращает сведения о том, является ли указанный год високосным
static DateTime Parse(...) **static** DateTime ParseExact(...) **static bool** TryParse(...) **static bool** TryParseExact(...)		Преобразовывают заданное строковое представление даты и времени в его эквивалент DateTime (см. § 3.2)
TimeSpan Subtract		Находит интервал между двумя

(DateTime value)		значениями даты и времени
DateTime Subtract (TimeSpan value)		Вычитает из этого экземпляра указанный интервал
override string ToString()	Object	Преобразует значение текущего объекта DateTime в эквивалентное ему строковое представление
string ToString(**string** format)		То же, но в указанном формате (см. § 3.2)
string ToString (IFormatProvider provider)	IConvertible	То же, но с использованием указанных сведений о форматировании, связанных с языком и региональными параметрами (см. п. 4.9.4)
string ToString(**string** format, IFormatProvider provider)	IFormattable	То же (см. п. 4.9.4)
DateTime ToUniversalTime()		Преобразует значение текущего объекта DateTime во время UTC
<тип> To<тип> (IFormatProvider provider)	IConvertible	Преобразование в другие форматы (не все поддерживаются, см. п. 4.9.4)
Операторы		
static DateTime **operator** + (DateTime d, TimeSpan t)		Прибавляет к дате и времени указанный временной интервал
static bool operator == (DateTime d1, DateTime d2)		Определяет, равны ли два заданных экземпляра
static bool operator > (DateTime t1, DateTime t2)		Определяет, больше ли один экземпляр другого
static bool operator >= (DateTime t1, DateTime t2)		По аналогии
static bool operator		По аналогии

!= (DateTime d1, DateTime d2)		
static bool operator < (DateTime t1, DateTime t2)		По аналогии
static bool operator <= (DateTime t1, DateTime t2)		По аналогии
static TimeSpan **operator** - (DateTime d1, DateTime d2)		Находит интервал между двумя значениями даты и времени
static DateTime **operator** - (DateTime d, TimeSpan t)		Вычитает из этого экземпляра указанный интервал
Поля		
static readonly DateTime MaxValue		Представляет наибольшее значение типа DateTime
static readonly DateTime MinValue		Представляет минимально допустимое значение типа DateTime
Свойства		
DateTime Date		Возвращает компоненту даты
int Day		Возвращает день месяца
DayOfWeek DayOfWeek		Возвращает день недели(2)
int DayOfYear		Возвращает день года
int Hour		Возвращает компонент часа
DateTimeKind Kind		Возвращает тип времени(3)
int Millisecond		Возвращает компонент миллисекунд
int Minute		Возвращает компонент минуты
int Month		Возвращает компонент месяца
static DateTime Now		Возвращает объект DateTime, которому присвоены текущие дата и время
int Second		Возвращает компонент секунды
long Ticks		Возвращает число тактов
TimeSpan TimeOfDay		Возвращает время дня
static DateTime Today		Возвращает текущую дату

static DateTime UtcNow		Возвращает объект DateTime, которому присвоены текущие дата и время UTC(4)
int Year		Возвращает компонент года

Примечания:

(1) TimeSpan – структура, представляет интервал времени. Содержит не меньший набор членов, чем структура DateTime. Подробности можно узнать в справочной системе.

(2) DayOfWeek – перечисление дней недели, может принимать следующие значения: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

(3) DateTimeKind – перечисление типов представления времени, может принимать следующие значения:

• Unspecified – Представленное время не определено ни как местное, ни как время UTC;

• Utc – Представленное время является временем UTC;

• Local – Представленное время является местным.

(4) UTC (Universal Time Coordinated) – всемирное координированное время. См. http://ru.wikipedia.org/wiki/UTC.

Приложение Б. Объекты для работы со строками

Основные открытые (public) члены класса System.String представлены в таблице Б.1.

Табл. Б.1 – Члены класса String

Член	Владелец	Описание
Конструкторы		
`String(...)`		Ряд конструкторов для создания строки из массивов элементов типа **char** и **byte** (см. MSDN)
Методы		
object `Clone()`	`ICloneable`	Возвращает ссылку на данный экземпляр класса String
static int `Compare(`**string** `strA,` **string** `strB)`		Сравнивает два указанных объекта String. При strA < strB результат меньше нуля, при strA = strB – ноль, при strA > strB – больше нуля
static int `Compare(...)`		Еще большой набор методов для сравнения строк с дополнительными опциями (сравнение подстрок, учет регистра, региональных параметров)
static int `CompareOrdinal(...)`		Сравнивает два заданных объекта String, оценивая числовые значения соответствующих объектов
int `CompareTo(`**object** `value)`	`IComparable`	Реализация сравнения для интерфейса IComparable
static string `Concat(...)`		Ряд методов, соединяющих строковое представление аргументов в единую строку

bool Contains(**string** value)		Определяет, входит ли подстрока в данный экземпляр строки
static string Copy(**string** str)		Создает копию данного экземпляра строки
void CopyTo(**int** sourceIndex, **char**[] destination, **int** destinationIndex, **int** count)		Копирует в строку указанные элементы массива char
bool EndsWith(...)		Определяет, совпадает ли конец данного экземпляра строки с указанной строкой
override bool Equals(**object** obj)	Object	Сравнивает две строки
bool Equals(...) **static bool** Equals(...)		Ряд методов для сравнения строк с дополнительными опциями
static string Format(...)		Формирует строку из аргументов в указанном формате
CharEnumerator GetEnumerator()		Возвращает итератор строки (см. п. 4.9.4 и описание цикла foreach)
IEnumerator GetEnumerator()	IEnumerable	То же самое
Type GetType()	Object	Возвращает метакласс типа
TypeCode GetTypeCode()	IConvertible	Возвращает TypeCode для класса String
int IndexOf(...)		Ряд методов, возвращающих позицию первого вхождения подстроки или символа в данную строку
int IndexOfAny(**char**[] anyOf, ...)		Ряд методов, возвращающих позицию первого вхождения любого символа из набора в данную строку

string Insert(**int** startIndex, **string** value)		Вставляет подстроку в данную строку, и возвращает результат в виде новой строки
static bool IsNullOrEmpty(**string** value)		Указывает, имеет ли указанная строка значение **null** или Empty(1)
static bool IsNullOrWhiteSpace (**string** value)		Указывает, является ли указанная строка значением null, пустой строкой или строкой, состоящей только из пробельных символов(2)
static string Join(...)		Сцепляет аргументы метода, помещая между ними заданный разделитель
int LastIndexOf(...)		Ряд методов, возвращающих позицию последнего вхождения подстроки или символа в данную строку
int LastIndexOfAny (**char**[] anyOf, ...)		Ряд методов, возвращающих позицию последнего вхождения любого символа из набора в данную строку
string PadLeft(**int** totalWidth)		Возвращает новую строку, дополняющую данную слева пробелами до указанной длины
string PadLeft(**int** totalWidth, **char** paddingChar)		Возвращает новую строку, дополняющую данную слева любыми символами до указанной длины
string PadRight(**int** totalWidth)		Возвращает новую строку, дополняющую данную справа пробелами до указанной длины

string PadRight(**int** totalWidth, **char** paddingChar)		Возвращает новую строку, дополняющую данную справа любыми символами до указанной длины
string Remove(...)		Возвращает новую строку с удаленными из данной строки символами
string Replace(...)		Возвращает новую строку, в которой заменены все вхождения указанного символа или подстроки
string[] Split(...)		Разделяет данную строку на части и возвращает результат в виде массив строк
bool StartsWith(...)		Определяет, совпадает ли начало данного экземпляра строки с указанной строкой
string Substring(...)		Извлекает подстроку из данного экземпляра и возвращает в виде новой строки
char[] ToCharArray(...)		Копирует символы данного экземпляра в массив **char**
string ToLower(...)		Возвращает новую строку, переведя символы текущей строки в нижний регистр
override string ToString()	Object	Т.к. класс сам является строкой, то возвращает свою копию
string ToString (IFormatProvider provider)	IConvertible	То же, но с указанием формата
string ToUpper(...)		Возвращает новую строку, переведя символы текущей строки в верхний регистр
string Trim()		Возвращает новую строку,

		удалив из текущей строки все начальные и конечные пробельные символы
string Trim(**params char**[] trimChars)		То же, но удаляются символы из указанного набора
string TrimEnd(**params char**[] trimChars)		То же, но удаляются только конечные символы
string TrimStart(**params char**[] trimChars)		То же, но удаляются только начальные символы
<тип> To<тип> (IFormatProvider provider)	IConvertible	Преобразование в другие форматы (см. п. 4.9.4)
Операторы		
static bool operator == (**string** a, **string** b)		Определяет, равны ли значения двух указанных строк
static bool operator != (**string** a, **string** b)		Определяет, различаются ли значения двух указанных строк
static string operator + (**string** a, **string** b)		Производит конкатенацию двух строк и возвращает результат в виде новой строки(3)
Поля		
static readonly string Empty		Представляет пустую строку
Свойства		
char this[**int** index]		Индексатор(4), возвращающий символ в указанной позиции
int Length		Возвращает длину строки

Примечания:

(1) **public static readonly string** String.Empty – поле класса String, представляющее пустую строку.

(2) Пробельные символы (символы-разделители) – символы, для которых метод Char.IsWhiteSpace возвращает **true**.

[3] Если в классе перегружен оператор «+», то он позволяет использовать также и оператор «+=» (см. п. 4.7.3).

[4] Индексаторы – механизм, заменяющий в языке C# перегрузку оператора индексации «[]», реализованную в языке C++ (см. п. 4.5.2).

Основные открытые (**public**) члены класса System.Text.StringBuilder представлены в таблице Б.2.

Табл. Б.2 – Члены класса StringBuilder

Член	Владелец	Описание
Конструкторы		
`StringBuilder(...)`		Ряд конструкторов для создания строки (см. MSDN)
Методы		
`StringBuilder Append(...)`		Добавляет строковое представление аргумента в конец данного экземпляра строки
`StringBuilder AppendFormat(...)`		То же, но с указанием формата
`void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)`		Аналогично методу String.CopyTo
`int EnsureCapacity(int capacity)`		Если требуемая емкость объекта больше текущей, увеличивает ее как минимум до указанного значения
`override bool Equals(object obj)`	`Object`	Сравнивает строку с текущей строкой
`bool Equals(StringBuilder sb)`		Сравнивает не только содержимое строк, но и емкость
`Type GetType()`	`Object`	Возвращает метакласс типа
`StringBuilder Insert(...)`		Вставляет строковое представление аргумента в текущую строку
`StringBuilder Remove(int startIndex, int length)`		Удаляет из текущей строки указанное количество символов

`StringBuilder Replace(...)`		Замещает в строке выбранные символы или подстроки
`override string` `ToString()`	`Object`	Преобразует объект в экземпляр класса String
`string` `ToString(`**`int`** `startIndex,` **`int`** `length)`		Преобразует часть символов объекта в экземпляр класса String
Свойства		
`int` `Capacity`		Дает доступ к чтению и модификации емкости объекта
`char this``[`**`int`** `index]`		Предоставляет доступ к символу по указанной позиции
`int` `Length`		Доступ к длине строки
`int` `MaxCapacity`		Возвращает максимально возможную емкость объекта

Приложение В. Объекты для работы с массивами

Основные открытые (**public**) члены класса System.Array представлены в таблице Б.1.

Табл. Б.1 – Члены класса Array

Член	Владелец	Описание
Методы		
static int BinarySearch(...)		Выполняет бинарный поиск заданного элемента во всем отсортированном по одной размерности массиве
static void Clear(Array array, **int** index, **int** length)		Задает диапазон элементов массива равным 0, **false** или **null** (в зависимости от типа элемента)
object Clone()	ICloneable	Создает неполную копию массива(1)
static void ConstrainedCopy(Array sourceArray, **int** sourceIndex, Array destinationArray, **int** destinationIndex, **int** length)		Копирует диапазон элементов из одного массива в другой. Если успешное завершение копирования невозможно, все изменения буду отменены
static TOutput[] ConvertAll <TInput, TOutput>(TInput[] array, Converter <TInput, TOutput> converter)		Преобразует массив одного типа в массив другого типа
static void Copy(...)		Копирует диапазон элементов из одного массива в другой
void CopyTo(Array array, **int** index)	ICollection	Копирует все элементы текущего одномерного массива в заданный одномерный массив

void CopyTo(Array array, **long** index)		То же
static Array CreateInstance (Type elementType, ...)		Инициализирует новый экземпляр класса Array
override bool Equals(**object** obj)	Object	Проверяет равенство массивов
static bool Exists<T>(T[] array, Predicate<T> match)		Определяет, содержит ли заданный массив элементы, удовлетворяющие условиям указанного предиката
static T Find<T>(T[] array, Predicate<T> match)		Выполняет поиск первого элемента, удовлетворяющего условиям указанного предиката
static T[] FindAll<T>(T[] array, Predicate<T> match)		Извлекает все элементы, удовлетворяющие условиям указанного предиката
static int FindIndex<T>(T[] array, ..., Predicate<T> match)		Выполняет поиск индекса первого элемента, удовлетворяющего условиям указанного предиката
static T FindLast<T>(T[] array, Predicate<T> match)		Выполняет поиск последнего элемента, удовлетворяющего условиям указанного предиката
static int FindLastIndex<T>(T[] array, ..., Predicate<T> match)		Выполняет поиск индекса последнего элемента, удовлетворяющего условиям указанного предиката
static void ForEach<T>(T[] array, Action<T> action)		Выполняет указанное действие для каждого элемента массива
IEnumerator GetEnumerator()	IEnumerable	Возвращает для массива интерфейс IEnumerator

int GetLength(**int** dimension) **long** GetLongLength(**int** dimension)		Возвращают количество элементов в заданном измерении массива
int GetLowerBound(**int** dimension)		Возвращает нижнюю границу заданного измерения массива
Type GetType()	Object	Возвращает метакласс типа
int GetUpperBound(**int** dimension)		Возвращает верхнюю границу заданного измерения массива
object GetValue(...)		Возвращает значение заданного элемента в текущем массиве
static int IndexOf(...)		Возвращает индекс первого вхождения указанного значения в одномерный массив
void Initialize()		Инициализирует каждый элемент массива типа значения путем вызова конструктора по умолчанию
static int LastIndexOf(...)		Возвращает индекс последнего вхождения указанного значения в одномерный массив
static void Resize<T>(**ref** T[] array, **int** newSize)		Заменяет размер массива указанным новым размером
static void Reverse(Array array)		Изменяет порядок элементов в одномерном массиве на обратный
void SetValue(**object** value, ...)		Присваивает элементу текущего массива заданное значение
static void Sort(...)		Сортирует элементы в одномерных массивах

Свойства		
bool IsFixedSize	IList	Возвращает значение, позволяющее определить, имеет ли массив фиксированный размер
bool IsReadOnly	IList	Возвращает значение, позволяющее определить, доступен ли массив только для чтения
bool IsSynchronized	ICollection	Возвращает значение, позволяющее определить, является ли доступ массиву синхронизированным(2)
int Length **long** LongLength		Возвращают общее число элементов во всех измерениях массива
int Rank		Возвращает размерность массива
object SyncRoot	ICollection	Возвращает объект, который можно использовать для синхронизации доступа к массиву(2)

Примечания:

(1) При неполном копировании массива копируются только элементы, имеющие ссылочный тип или тип значения. Объекты, на которые указывают ссылки, не копируются. Ссылки в новом массиве указывают на те же объекты, что и ссылки в исходном массиве.

(2) Подробнее о синхронизации в п. 5.2.3.

Приложение Г. Объекты форматирования

Табл. Г.1 – Основные члены класса System.Globalization.NumberFormatInfo

Член	Описание
Конструкторы	
`NumberFormatInfo()`	Инициализирует новый экземпляр класса NumberFormatInfo, не зависящий от языка и региональных параметров (инвариантный)
Свойства	
int `CurrencyDecimalDigits`	Число десятичных разрядов, используемое в значениях денежных единиц (0-99)
string `CurrencyDecimalSeparator`	Строка, используемая в качестве десятичного разделителя в значениях денежных единиц
string `CurrencyGroupSeparator`	Строка, разделяющая разряды в целой части значений денежных единиц
int`[]` `CurrencyGroupSizes`	Число цифр в каждой из групп целой части значений денежных единиц
int `CurrencyNegativePattern`	Шаблон формата для отрицательных значений денежных единиц (0-15)
int `CurrencyPositivePattern`	Шаблон формата для положительных значений денежных единиц (0-3)
string `CurrencySymbol`	Строка, используемая в качестве знака денежной единицы
string `NaNSymbol`	Строка, представляющая значение IEEE NaN(1)
string`[]` `NativeDigits`	Массив строк собственных цифр, эквивалентных арабским цифрам от 0 до 9
string `NegativeInfinitySymbol`	Строка, представляющая минус бесконечность
string `NegativeSign`	Строка, указывающая, что соответствующее число является отрицательным
int `NumberDecimalDigits`	Число десятичных разрядов, используемое в числовых значениях

string NumberDecimalSeparator	Строка, используемая в качестве десятичного разделителя в числовых значениях
string NumberGroupSeparator	Строка, разделяющая разряды в целой части десятичной дроби в числовых значениях
int[] NumberGroupSizes	Число цифр в каждой из групп целой части десятичной дроби в числовых значениях
int NumberNegativePattern	Шаблон формата для отрицательных числовых значений (0-4)
int PercentDecimalDigits	Число десятичных разрядов, используемое в значениях процентов
string PercentDecimalSeparator	Строка, используемая в качестве десятичного разделителя в значениях процентов
string PercentGroupSeparator	Строка, разделяющая разряды в целой части десятичной дроби в значениях процентов
int[] PercentGroupSizes	Число цифр в каждой из групп целой части десятичной дроби в значениях процентов
int PercentNegativePattern	Шаблон формата для отрицательных значений процентов (0-11)
int PercentPositivePattern	Шаблон формата для положительных значений процентов (0-3)
string PercentSymbol	Строка, используемая в качестве знака процентов
string PerMilleSymbol	Строка, используемая в качестве знака промилле(2)
string PositiveInfinitySymbol	Строка, представляющая плюс бесконечность
string PositiveSign	Строка, указывающая, что соответствующее число является положительным

Примечания:

(1) NaN (not a number, не число) – особое состояние числа с плавающей точкой. Обычно является результатом ошибки предыдущей математической операции (деление ноля на ноль, вычисление корня из отрицательного числа и т.д.).

(2) Промилле – одна тысячная доля, или 1/10 процента. По умолчанию

для инвариантной культуры обозначается символом U+2030 (‰).

Табл. Г.2 – Основные члены класса System.Globalization.DateTimeFormatInfo

Член	Описание
Конструкторы	
`DateTimeFormatInfo()`	Инициализирует новый экземпляр класса DateTimeFormatInfo, не зависящий от языка и региональных параметров (инвариантный)
Методы	
`string` `GetAbbreviatedDayName(DayOfWeek dayofweek)`	Возвращает полное название указанного дня недели, принятое в определенном языке и региональных параметрах
`string` `GetAbbreviatedEraName(`**`int`** `era)`	Возвращает строку, содержащую сокращенное название указанной эры
`string` `GetAbbreviatedMonthName(`**`int`** `month)`	Возвращает сокращение указанного месяца, принятое в определенном языке и региональных параметрах
`string``[] GetAllDateTimePatterns()`	Возвращает все стандартные шаблоны, в которых можно форматировать значения DateTime
`string``[] GetAllDateTimePatterns(`**`char`** `format)`	Возвращает все стандартные шаблоны, в которых можно форматировать значения DateTime с помощью указанного шаблона формата
`string` `GetDayName(DayOfWeek dayofweek)`	Возвращает сокращение указанного дня недели, принятое в определенном языке и региональных параметрах
`int` `GetEra(`**`string`** `eraName)`	Возвращает целое число, пред-

	ставляющее собой указанную эру
string GetEraName(**int** era)	Возвращает строку, содержащую название указанной эры
string GetMonthName(**int** month)	Возвращает полное имя указанного месяца, принятое в определенном языке и региональных параметрах
string GetShortestDayName(DayOfWeek dayOfWeek)	Получает самое короткое сокращенное название дня для определенного дня недели
void SetAllDateTimePatterns(**string**[] patterns, **char** format)	Устанавливает стандартные шаблоны, в которых можно форматировать значения DateTime
Свойства	
string[] AbbreviatedDayNames	Сокращения дней недели, принятые в определенном языке и региональных параметрах
string[] AbbreviatedMonthNames	Сокращения месяцев, принятые в определенном языке и региональных параметрах
string AMDesignator	Строка указателя часов до полудня (AM – ante meridiem)
string DateSeparator	Строка, разделяющая компоненты даты – год, месяц и день
string[] DayNames	Полные названия дней недели, принятые в определенном языке и региональных параметрах
DayOfWeek FirstDayOfWeek	Первый день недели
string FullDateTimePattern	Шаблон формата для длинного формата даты и длинного формата времени, связанный с шаблоном формата «F»
string LongDatePattern	Шаблон формата для длинного формата даты и длинного формата времени, связанный с шабло-

	ном формата «D»
string LongTimePattern	Шаблон формата для длинного формата времени, связанного с шаблоном формата «T»
string MonthDayPattern	Шаблон формата для значения месяца и дня, связанный с шаблонами формата «m» и «M»
string[] MonthNames	Полные названия месяцев, принятые в определенном языке и региональных параметрах
string PMDesignator	Строка указателя часов после полудня (PM – post meridiem)
string RFC1123Pattern	Шаблон формата для значения времени на основе спецификации 1123 RFC IETF, связанный с шаблонами формата «r» и «R»
string ShortDatePattern	Шаблон формата для краткого формата даты, связанный с шаблоном формата «d»
string[] ShortestDayNames	Самые краткие уникальные сокращения имен дней
string ShortTimePattern	Шаблон формата для краткого формата времени, связанного с шаблоном формата «t»
string SortableDateTimePattern	Шаблон формата для значения даты и времени, допускающего сортировку, связанный с шаблоном формата «s»
string TimeSeparator	Строка, разделяющая компоненты времени – часы, минуты и секунды
string UniversalSortableDateTimePattern	Шаблон формата для значения всемирной даты и времени, допускающего сортировку, связанный с шаблоном формата «u»

string YearMonthPattern	Шаблон формата для значения месяца и дня, связанный с шаблонами формата «y» и «Y»

Табл. Г.3 – Основные члены класса System.Globalization.CultureInfo

Член	Описание
Конструкторы	
CultureInfo(**int** culture)	Инициализирует новый экземпляр класса CultureInfo на основе языка и региональных параметров, заданных идентификатором
CultureInfo(**string** name)	Инициализирует новый экземпляр класса CultureInfo, на основе языка и региональных параметров, заданных именем
Методы	
static CultureInfo CreateSpecificCulture(**string** name)	Создает объект CultureInfo, который представляет определенный язык и региональные параметры, соответствующие заданному имени
static CultureInfo[] GetCultures(CultureTypes types)	Возвращает список поддерживаемых языков и региональных параметров
Свойства	
CultureTypes CultureTypes	Возвращает типы языков и региональных параметров
static CultureInfo CurrentCulture	Возвращает CultureInfo, представляющий язык и региональные параметры, используемые текущим потоком
DateTimeFormatInfo DateTimeFormat	Объект DateTimeFormatInfo, определяющий формат отображения даты и времени, соответствующий языку и региональным параметрам
static CultureInfo InstalledUICulture	Возвращает значение CultureInfo, представляющее язык и региональные параметры, установленные с операци-

	онной системой
static CultureInfo InvariantCulture	Возвращает объект CultureInfo, не зависящий от языка и региональных параметров (инвариантный)
bool IsNeutralCulture	Возвращает значение, показывающее, представляет ли текущий CultureInfo нейтральный язык и региональные параметры
int KeyboardLayoutId	Получает активный идентификатор языка ввода (раскладку клавиатуры)
int LCID	Возвращает идентификатор языка и региональных параметров
string Name	Возвращает имя языка и региональных параметров
string NativeName	Получает имя языка и региональных параметров, состоящих из языка, страны или региона и дополнительного набора символов, которые свойственны этому языку
NumberFormatInfo NumberFormat	Объект NumberFormatInfo, определяющий формат отображения чисел, денежной единицы и процентов, соответствующий языку и региональным параметрам

Приложение Д. Объекты файлового ввода-вывода

Объекты для бинарного потокового ввода-вывода рассмотрены в табл. Д.1-Д.6.

Табл. Д.1 – Основные члены класса System.IO.Stream

Член	Владелец	Описание
Методы		
virtual void Close()		Закрывает текущий поток
void Dispose()	IDisposable	Удаление потока. Все изменения, накопленные в буфере, записываются в поток, а затем поток закрывается. Напрямую вызывать данный метод не следует, он вызывается автоматически.
virtual void Dispose(**bool** disposing)		Если параметр равен false, освобождает неуправляемые ресурсы потока, если true – освобождает все ресурсы
abstract void Flush()		Очищает все буферы данного потока и вызывает запись данных буферов в базовое устройство
abstract int Read(**byte**[] buffer, **int** offset, **int** count)		Считывает последовательность байтов из текущего потока и перемещает позицию в потоке на число считанных байтов
virtual int ReadByte()		Считывает байт из потока и перемещает позицию в потоке на один байт, или возвращает –1, если достигнут конец потока
abstract long Seek(**long** offset, SeekOrigin origin)		Задает позицию в текущем потоке(1)
abstract void SetLength(**long** value)		Задает длину текущего потока. Поддерживается не для всех типов потоков.

abstract void Write(**byte**[] buffer, **int** offset, **int** count)		Записывает последовательность байтов в текущий поток и перемещает текущую позицию в нем вперед на число записанных байтов
virtual void WriteByte(**byte** value)		Записывает байт в текущее положение в потоке и перемещает позицию в потоке вперед на один байт
Поля		
static readonly Stream Null		Класс Stream без резервного хранилища(2)
Свойства		
abstract bool CanRead		Показывает, поддерживает ли текущий поток возможность чтения
abstract bool CanSeek		Показывает, поддерживается ли в текущем потоке возможность перемещения
abstract bool CanWrite		Показывает, поддерживает ли текущий поток возможность записи
abstract long Length		Длина потока в байтах
abstract long Position		Получает или задает (если это разрешено) позицию в текущем потоке

Примечания:

(1) Перечисление System.IO.SeekOrigin содержит три константы для типа смещения – Begin (относительно начала потока), Current (относительно текущей позиции) и End (относительно конца потока).

(2) Запись в поток Null приводит к тому, что данные не записываются. Прочитать из него тоже ничего нельзя. Нужен он в тех случаях, когда необходимо переправить куда-то поток постоянно поступающих данных (например, из сетевого канала), не затрачивая ресурсы операционной системы. Аналог – перенаправление вывода в командной строке в виртуальный файл nul или запись данных в этот файл, например,

```
copy 1.txt nul
copy 1.txt 2.txt > nul
```

и т.д.

Для перечисленных ниже классов-потомков IO.Stream (табл. Д.2-Д.4) указаны только новые члены, отсутствующие в базовом классе, или переопределенные члены, чье поведение существенно отличается от базовой функциональности.

Табл. Д.2 – Основные члены класса System.IO.Compression.GZipStream

Член	Владелец	Описание
Конструкторы		
`GZipStream(Stream stream, CompressionMode mode)`		Инициализирует новый экземпляр класса с использованием заданного базового потока и режима сжатия(1)
`GZipStream(Stream stream, CompressionMode mode, bool leaveOpen)`		То же самое, но с указанием – следует ли при закрытии потока GZip оставить базовый поток открытым
Методы		
`override void SetLength(long value)`	`Stream`	Это свойство не поддерживается и всегда вызывает исключительную ситуацию NotSupportedException
Свойства		
`Stream BaseStream`		Возвращает ссылку на базовый поток
`override long Length`	`Stream`	Это свойство не поддерживается и всегда вызывает исключительную ситуацию NotSupportedException
`override long Position`	`Stream`	Аналогично

Примечания:

(1) Режим сжатия System.IO.Compression.CompressionMode содержит две константы – Decompress (распаковка) и Compress (упаковка).

Табл. Д.3 – Основные члены класса System.IO.FileStream

Член	Владелец	Описание
Конструкторы		
`FileStream(string path, FileMode mode, FileAccess access)`		Инициализирует новый экземпляр класса с указанным путем, режимом создания(1) и режимом доступа(2)

`FileStream(...)`		Большой набор других конструкторов (см. MSDN)
Методы		
virtual void `Lock(`**long** `position,` **long** `length)`		Предотвращает изменение потока другими процессами, пока разрешен доступ для чтения
virtual void `Unlock(`**long** `position,` **long** `length)`		Разрешает доступ других процессов ко всему ранее заблокированному файлу или его части
Свойства		
string `Name`		Возвращает передаваемое конструктору имя файла

Примечания:

(1) Режим создания System.IO.FileMode содержит следующие константы:

• CreateNew – создается новый файл (если такой файл уже существует, создается исключение IOException);

• Create – создается новый файл (если такой файл уже существует, то он будет перезаписан);

• Open – открывается существующий файл (если файл не существует, создается исключение FileNotFoundException);

• OpenOrCreate – файл открывается, если он существует, в противном случае создается новый файл;

• Truncate – открывается существующий файл, после чего его содержимое удаляется, т.е. размер становится равен нулю;

• Append – если файл существует, то он открывается, и текущей становится позиция в конце файла, иначе создается новый файл.

(2) Режим доступа System.IO.FileAccess содержит следующие константы:

• Read – доступ для чтения файла;

• Write – доступ для записи файла;

• ReadWrite – доступ для чтения и записи.

Табл. Д.4 – Основные члены класса System.IO.MemoryStream

Член	Владелец	Описание

Конструкторы		
`MemoryStream(...)`		Большой набор конструкторов (см. MSDN)
Методы		
virtual byte[] GetBuffer()		Возвращает массив байтов, из которого был создан данный поток
virtual byte[] ToArray()		Записывает содержимое всего потока в массив байтов
virtual void WriteTo(Stream stream)		Записывает все содержимое данного потока памяти в другой поток
Свойства		
virtual int Capacity		Число байтов, выделенных для этого потока

Табл. Д.5 – Основные члены класса System.IO.BinaryReader

Член	Владелец	Описание
Конструкторы		
`BinaryReader(Stream input)`		Выполняет инициализацию нового экземпляра класса с базовым потоком и кодировкой UTF8Encoding
`BinaryReader(Stream input, Encoding encoding)`		Выполняет инициализацию нового экземпляра класса с базовым потоком и указанной кодировкой(1)
Методы		
virtual void Close()		Закрывает текущий поток чтения и связанный с ним базовый поток
virtual void Dispose(**bool** disposing)		Если параметр равен false, освобождает неуправляемые ресурсы потока, если true – освобождает все ресурсы
virtual void FillBuffer(**int** numBytes)		Заполняет внутренний буфер указанным количеством байтов, которые были считаны из потока
virtual int PeekChar()		Возвращает следующий доступный для чтения символ, не перемещая

		позицию байта или символа вперед
virtual <тип> Read...(...)		Большой набор методов для чтения из потока данных и массивов различных типов (см. MSDN)
Свойства		
virtual Stream BaseStream		Предоставляет доступ к базовому потоку

Примечания:

(1) System.Text.Encoding – абстрактный базовый класс для других классов, представляющих различные кодировки символов (ASCIIEncoding, UnicodeEncoding, UTF32Encoding, UTF7Encoding, UTF8Encoding). Подробности см. в MSDN.

Табл. Д.6 – Основные члены класса System.IO.BinaryWriter

Член	Владелец	Описание
Конструкторы		
BinaryWriter(Stream output)		Выполняет инициализацию нового экземпляра класса с базовым потоком и кодировкой UTF8Encoding
BinaryWriter(Stream output, Encoding encoding)		Выполняет инициализацию нового экземпляра класса с базовым потоком и указанной кодировкой
Методы		
virtual void Close()		Закрывает текущий поток записи и связанный с ним базовый поток
virtual void Dispose(**bool** disposing)		Если параметр равен false, освобождает неуправляемые ресурсы потока, если true – освобождает все ресурсы
virtual void Flush()		Очищает все буферы текущего модуля записи и вызывает немедленную запись всех буферизованных данных на базовое устройство
virtual long Seek(**int** offset, SeekOrigin		Устанавливает позицию в текущем потоке

origin)		
virtual void Write...(...)		Большой набор методов для записи в поток данных и массивов различных типов (см. MSDN)
Поля		
static readonly BinaryWriter Null		Возвращает поток без резервного хранилища
Свойства		
virtual Stream BaseStream		Предоставляет доступ к базовому потоку

Объекты для текстового потокового ввода-вывода рассмотрены в табл. Д.7-Д.12.

Табл. Д.7 – Основные члены класса System.IO.TextReader

Член	Владелец	Описание
Методы		
virtual void Close()		Закрывает поток и освобождает все системные ресурсы, связанные с ним
void Dispose()	IDisposable	Освобождает все ресурсы, используемые объектом
virtual void Dispose(**bool** disposing)		Если параметр равен false, освобождает неуправляемые ресурсы потока, если true – освобождает все ресурсы
virtual int Peek()		Возвращает следующий доступный символ, фактически не считывая его из потока входных данных (т.е. не изменяя текущую позицию потока)
virtual int Read()		Выполняет чтение следующего символа из входного потока и перемещает положение символа на одну позицию вперед
virtual int		Выполняет чтение диапазона

`ReadBlock(char[] buffer, int index, int count)`		символов в буфер
`virtual string ReadLine()`		Выполняет чтение строки символов из текущего потока
`virtual string ReadToEnd()`		Выполняет чтение всех символов, начиная с текущей позиции до конца потока
Поля		
`static readonly TextReader Null`		Предоставляет поток без данных, доступных для чтения

Табл. Д.8 – Основные члены класса System.IO.TextWriter

Член	Владелец	Описание
Методы		
`virtual void Close()`		Закрывает поток и освобождает все системные ресурсы, связанные с ним
`void Dispose()`	`IDisposable`	Освобождает все ресурсы, используемые объектом
`virtual void Dispose(bool disposing)`		Если параметр равен false, освобождает неуправляемые ресурсы потока, если true – освобождает все ресурсы
`virtual void Flush()`		Очищает все буферы текущего потока и вызывает немедленную запись всех буферизованных данных на базовое устройство
`virtual void Write(...)`		Большой набор методов для записи в поток переменных и массивов различных видов (аналогично методу Console.Write)
`virtual void WriteLine(...)`		То же самое, но строка записывается с признаком конца строки (аналогично методу Console.WriteLine)
Поля		
`static readonly TextWriter Null`		Предоставляет поток без резервного хранилища, в который можно осу-

		ществлять запись, но из которого нельзя считывать данные
Свойства		
abstract Encoding Encoding		Возвращает кодировку, в которой записаны выходные данные
virtual IFormatProvider FormatProvider		Возвращает объект, управляющий форматированием
virtual string NewLine		Признак конца строки, используемый текущим потоком

Для перечисленных ниже (табл. Д.9-Д.12) классов, являющихся потомками IO.TextReader и IO.TextWriter указаны только новые члены, отсутствующие в базовом классе.

Табл. Д.9 – Основные члены класса System.IO.StreamReader

Член	Владелец	Описание
Конструкторы		
StreamReader(**string** path)		Инициализирует новый экземпляр потока для указанного имени файла
StreamReader(Stream stream)		Инициализирует новый экземпляр класса для указанного базового потока
StreamReader(...)		Ряд других конструкторов (см. MSDN)
Поля		
virtual Stream BaseStream		Возвращает основной поток
virtual Encoding CurrentEncoding		Возвращает текущую кодировку символов потока
bool EndOfStream		Определяет, находится ли позиция текущего потока в конце потока

Табл. Д.10 – Основные члены класса System.IO.StringReader

Член	Владелец	Описание
Конструкторы		
StringReader(**string** s)		Инициализирует новый экземпляр пото-

		ка, осуществляющий чтение из указанной строки

Табл. Д.11 – Основные члены класса System.IO.StreamWriter

Член	Владелец	Описание
Конструкторы		
`StreamWriter(`**`string`** `path)`		Инициализирует новый экземпляр потока для указанного имени файла
`StreamWriter(Stream stream)`		Инициализирует новый экземпляр класса для указанного базового потока
`StreamWriter(...)`		Ряд других конструкторов (см. MSDN)
Поля		
`virtual bool` `AutoFlush`		Определяет, будет ли сбрасываться буфер в основной поток после каждого вызова Write и WriteLine
`virtual` `Stream BaseStream`		Возвращает основной поток

Табл. Д.12 – Основные члены класса System.IO.StringWriter

Член	Владелец	Описание
Конструкторы		
`StringWriter(...)`		Набор конструкторов для инициализации новых экземпляров потока