1 Основы Microsoft .NET

1.1 Краткие исторические сведения

Разработка платформы .NET (читается «дот нет») началась в 1998 году. Первоначально ей дали название Project 42, которое затем было изменено на COM Object Runtime (COR). Аббревиатура COR использовалась достаточно долго — ее до сих пор можно найти в названиях dll-файлов и именах библиотечных функций. Потом платформа сменила еще несколько названий: Lightning, COM+ 2.0, Next Generation Web Services и, в конце концов, стала называться .NET Framework.

Многие идеи, воплощенные в .NET, появились задолго до ее появления на свет. В 1978 году в Калифорнийском университете для учебных целей была разработана операционная система UCSD p-System. Главное ее достоинство заключалось в том, что она могла работать как на компьютерах PDP-11 вычислительного центра университета, так и на микрокомпьютерах студентов.

Независимость операционной системы от аппаратной платформы достигалась путем введения понятия виртуальной p-машины (p-Machine), обладавшей собственным набором инструкций, который назывался p-кодом (p-code). Сама операционная система и все работавшие в ней программы были закодированы p-кодом, поэтому для того чтобы запустить их на новой аппаратной платформе, требовалось всего лишь реализовать для этой платформы интерпретатор p-кода.

Основное отличие UCSD p-System от .NET заключается в принципах выполнения программ. Программы, закодированные в p-коде, непосредственно выполнялись интерпретатором, тогда как программы на CIL (Common Intermediate Language) перед выполнением транслируются в код конкретного процессора специальным компилятором.

Технология ANDF (Architectural Neutral Distribution Format) была разработана в первой половине 1990-х годов в OSF (Open Software Foundation) для увеличения переносимости программного обеспечения. Смысл технологии заключается в разделении процесса компиляции программ на две разнесенные во времени и пространстве фазы:

1) перевод программы в формат ANDF;

2) трансляция программы, представленной в формате ANDF, в исполняемый файл при установке программы на компьютер пользователя.

Формат ANDF не зависит ни от языков программирования, ни от особенностей аппаратных платформ и операционных систем. Программы, распространяемые в формате ANDF, могут быть установлены на любой платформе, для которой имеется транслятор из ANDF в исполняемый код.

Платформа Java наиболее близка к платформе .NET по архитектуре и своим возможностям. Она была разработана в середине 1990-х годов в Sun Microsystems для бытовых приборов, подключаемых к компьютерным сетям. Стремительное развитие технологий Internet способствовало распространению платформы Java, которая является конкурентом .NET.

Краеугольным камнем платформы Java является виртуальная машина, которая отвечает за независимость Java-программ от операционных систем и аппаратных платформ. Набор инструкций этой виртуальной машины (Java byte-code) может выполняться как на специализированных Java-процессорах, так и путем компиляции в исполняемый код конкретной аппаратной платформы.

1.2 Достоинства .NET

Система программирования (платформа) .NET является наиболее значительным достижением корпорации Microsoft. Она воплотила лучшие идеи объектно-ориентированного программирования, технологий создания переносимых двоичных компонентов, распределенных приложений, межъязыкового взаимодействия и многое другое.

Достоинствами платформы .NET являются:

- возможность создания безопасных переносимых двоичных программных компонентов, которые могут исполняться на любой платформе, на которой установлена .NET Framework; код, компилируемый для .NET Framework, называется управляемым (managed code);

- полное взаимодействие с существующим кодом; код, написанный при помощи других программных средств или для взаимодействия с таким кодом, называется неуправляемым (unmanaged code);

- полное и абсолютное межъязыковое взаимодействие; в .NET поддерживаются межъязыковые наследование, обработка исключений и отладка; программные компоненты могут быть написаны одновременно на нескольких языках .NET;

- общая среда выполнения любых программных компонентов .NET, и, что представляется особенно важным, общие типы данных;

- общая библиотека базовых типов (классов), обеспечивающая целостную объектную модель языков .NET, что помогает легко ориентироваться в программах, написанных на любом языке .NET;

- упрощение развертывания приложений .NET; отсутствует регистрация типов в системном реестре, организована поддержка функционирования программных компонентов разных версий одновременно;

- действительно упрощенная работа со строковыми типами данных; нет никакого многообразия типов, внедрена поддержка Unicode не только в строковых переменных, но и в самой среде разработки.

1.3 Архитектура .NET Framework

Основу .NET Framework составляет CLR (Common Language Runtime, — стандартная общая среда выполнения) и .NET Framework class library (библиотека базовых типов, рисунок 1).

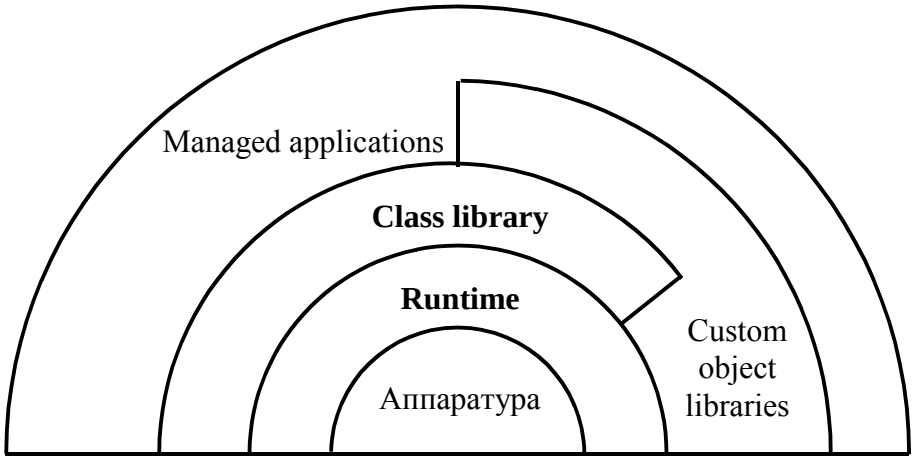

Рисунок 1 — Отношение между CLR и библиотекой типов

CLR обеспечивает обнаружение и загрузку типов .NET, их компиляцию, связывание и выполнение. CLR управляет памятью, потоками, межъязыковым взаимодействием, отслеживанием версий и т.п.

Библиотека базовых типов предоставляет многоуровневые функции для работы с потоками, файлами, данными, графикой, XML, SOAP и т.п. Библиотека типов организована в виде иерархии пространств имен и типов, которые позволяют легко и просто решать задачи, возникающие при программировании приложений самого разного назначения.

CLR основана на CTS и CLS (рисунок 2).

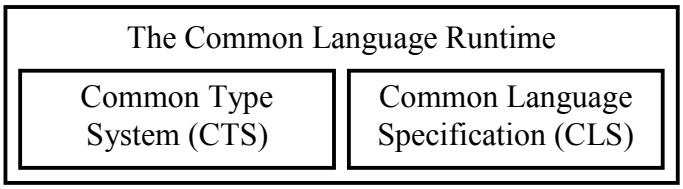

Рисунок 2 — CLR,CTS,CLS

CTS (Common Type System, стандартная система типов) обеспечивает базовые типы данных, поддерживаемых CLR, а также определяет, как одни типы могут взаимодействовать с другими типами и как они будут представлены в форме метаданных .NET.

CLS (Common Language Specification, стандартная спецификация языка) — набор правил, определяющих подмножество общих типов данных, в отношении которых гарантируется, что они безопасны при использовании во всех языках .NET.

1.4 Компиляция программ .NET

Платформа .NET использует двойную компиляцию исходного кода с тем, чтобы получить исполняемый код приложения (рисунок 3).

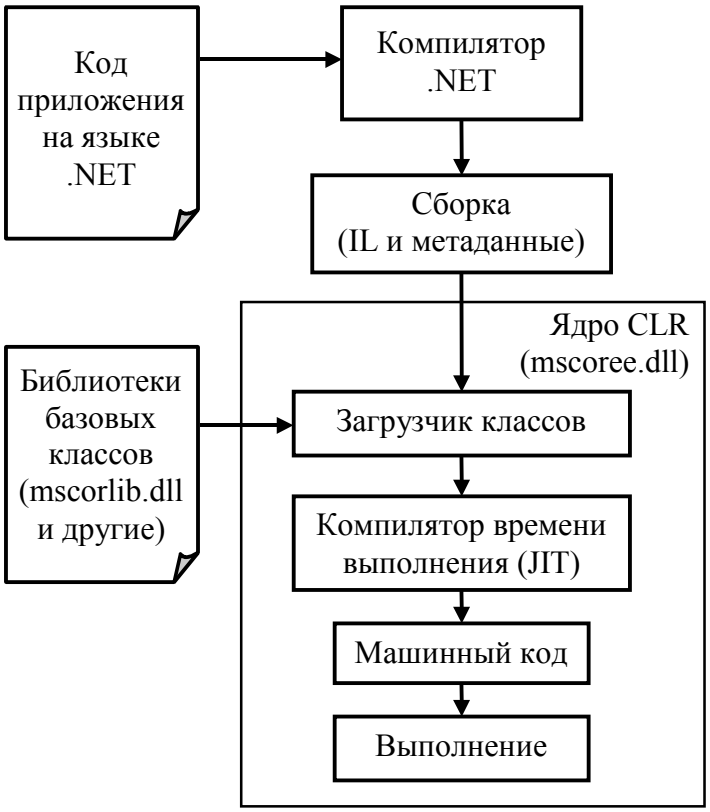

Рисунок 3 — Компиляция в .NET

Код программы при помощи одного из компиляторов .NET компилируется в платформенно-независимый промежуточный язык MSIL или просто IL (Microsoft Intermediate Language). Результатом этой компиляции является файл типа exe или dll, называемый сборкой (assembly). Сборка содержит инструкции IL и так называемые метаданные.

CLR загружает сборку, находит необходимые для ее выполнения типы, компилирует и выполняет машинный код.

Компиляцию в машинно-зависимые инструкции производит компилятор времени выполнения JIT (just-in-time-compiler). Компиляция производится только для вызываемого метода, а скомпилированный код помещается в кэш, что значительно ускоряет работу приложения.

1.5 Сборки

Приложения .NET создаются путем объединения некоторого количества сборок. Сборка — это двоичный файл, который содержит в себе номер версии, метаданные и типы (классы, интерфейсы, структуры).

Сборки являются контейнерами для типов и ресурсов, обеспечивая их уникальность и исключая конфликты версий. Типы .NET идентифицируются в том числе и по сборке, их содержащей, поэтому приложение может использовать две сборки, в которых описаны типы с одинаковым названием.

Метаданные сборки соответствуют библиотеке типов COM, однако они более полны и точны. Метаданные генерируются автоматически.

Часть метаданных описывает саму сборку и называется ее манифестом (manifest). Манифест содержит информацию обо всех двоичных файлах сборки, номер версии сборки, сведения обо всех внешних сборках, которые необходимы для выполнения данной сборки. Можно сказать, что сборки являются самодокументированными.

Любая сборка может состоять из одного или нескольких модулей. Модуль — это двоичный файл сборки .NET. Структура сборки, состоящей из одного файла (модуля) приведена на рисунке 4.

Рисунок 4 — Структура одно-файловой сборки

В сборке из нескольких файлов манифест размещается в одном из файлов.

1.5.1 Частные сборки

Сборка .NET может быть либо частной (private), либо сборкой для общего доступа (shared). Различия между этими типами сборок заключаются в правилах их именования, политики версий и размещения сборок на компьютере.

Частные сборки — это наборы типов, которые могут быть использованы только теми приложениями, в состав которых они входят. Частные сборки находятся в каталоге приложения-владельца, или в его подкаталогах. Каталог приложения-владельца называется каталогом приложения (application directory). При этом можно свободно менять расположение каталога приложения. Для установки программы .NET достаточно лишь скопировать каталог приложения, а для удаления программы достаточно удалить этот каталог.

Для идентификации частной сборки используется дружественное имя и номер версии. Например, для некоторой сборки CarLibrary в манифесте может быть записано:

```
.assembly CarLibrary as "CarLibrary" {
    . . .
    .ver 1:0:0:1
}
```

Среда выполнения CLR не использует никакой политики в отношении версий частных сборок (в этом нет необходимости). Частная сборка всегда находится в своем месте в единственном экземпляре. Несколько версий частной сборки могут находиться в разных каталогах — это не вызывает никаких конфликтов.

1.5.2 Файлы конфигурации приложений

Файл конфигурации приложения — это текстовый файл в формате XML, имеющий то же имя, что и приложение, и расширение .config.

В файле конфигурации можно указать подкаталоги, в которых следует искать сборки приложения.

Например, если у нас есть сборка с именем CarLib.dll и приложение CarApp.exe, расположенное в каталоге CarApp, то по умолчанию файл CarLib.dll автоматически размещается в каталоге CarApp. Мы можем создать каталог CarLib внутри каталога CarApp и разместить в нем сборку CarLib.dll. В этом случае в каталоге CarApp необходимо разместить файл с именем CarApp.exe.config, описывающий частный путь:

```
<configuration>
    <runtime>
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <probing privatePath="CarLib" />
        </assemblyBinding>
    </runtime>
</configuration>
```

Здесь термин probing означает процесс поиска сборок.

1.5.3 Сборки общего доступа

Сборки общего доступа предназначены для использования любыми приложениями .NET. Сборка общего доступа устанавливается в специальный каталог, называемый Global Assembly Cache (GAC, глобальный кэш сборок). Расположение этого каталога — C:\Windows\assembly (каталог имеет атрибут «системный»).

В сборках общего доступа используется дополнительная информация — «сильное имя» (strong name). Сильное имя состоит из:

- дружественного имени, культурной информации (culture information) и номера версии;

- открытого ключа;

- цифровой подписи.

Создание сильного имени основывается на криптографии открытого ключа. При создании сборки общего доступа сначала создается пара «открытый/закрытый ключ». Маркер открытого ключа помещается в манифест сборки, закрытый ключ используется для создания цифровой подписи, которая также помещается в сборку. Закрытый ключ хранится в файле сборки, который содержит манифест.

Сильное имя гарантирует, что используется правильная сборка, и что сборка не была изменена.

Для генерации пары «открытый/закрытый ключ» используется утилита sn.exe. Пример ее использования:

```
sn -k keys.snk
```

Полученные ключи используются для подписи сборки. Для этого необходимо открыть свойства проекта, выбрать вкладку Signing, включить флажок «Sign the assembly» и указать путь к файлу ключей, после чего скомпилировать проект.

Для того, чтобы поместить сборку общего доступа в GAC, используется утилита gacutil.exe. Пример ее использования:

```
gacutil -i CarLib.dll
```

Второе важное отличие сборок общего доступа заключается в использовании определенной политики в отношении версии сборки. Версия сборки состоит из четырех частей:

```
<major version>.<minor version>.<build number>.<revision>
```

Пример номера версии: 1:0:0:1.

Если у двух сборок не совпадают первые два номера версии сборки, CLR считает их полностью несовместимыми.

Если у двух сборок совпадают первые два номера, но различаются третьи номера, CLR считает их частично совместимыми (более старшая совместима с более младшей).

Если у сборок не совпадают только последние номера, сборки считаются полностью совместимыми.

Эта политика в отношении версий сборок является политикой по умолчанию. Она может быть изменена при помощи файла конфигурации приложения и администраторского файла конфигурации.

1.5.4 Просмотр сборок

Для просмотра сборок можно использовать утилиту ILDasm, приложение WinCV или ObjectBrowser, входящий в комплект поставки Visual Studio.

ILDasm (Intermediate Language Disassembler, дизассемблер промежуточного языка) позволяет просмотреть содержимое любой сборки — манифест, метаданные и инструкции IL.

Например, сборка CarLib.dll описывает следующий класс:

```
namespace CarLibrary {
    public class Car {
        public Car() { }
        public void Move() { Console.WriteLine("Hello\n"); }
    }
}
```

Вид этой сборки в утилите ILDasm приведен на рисунке 5.

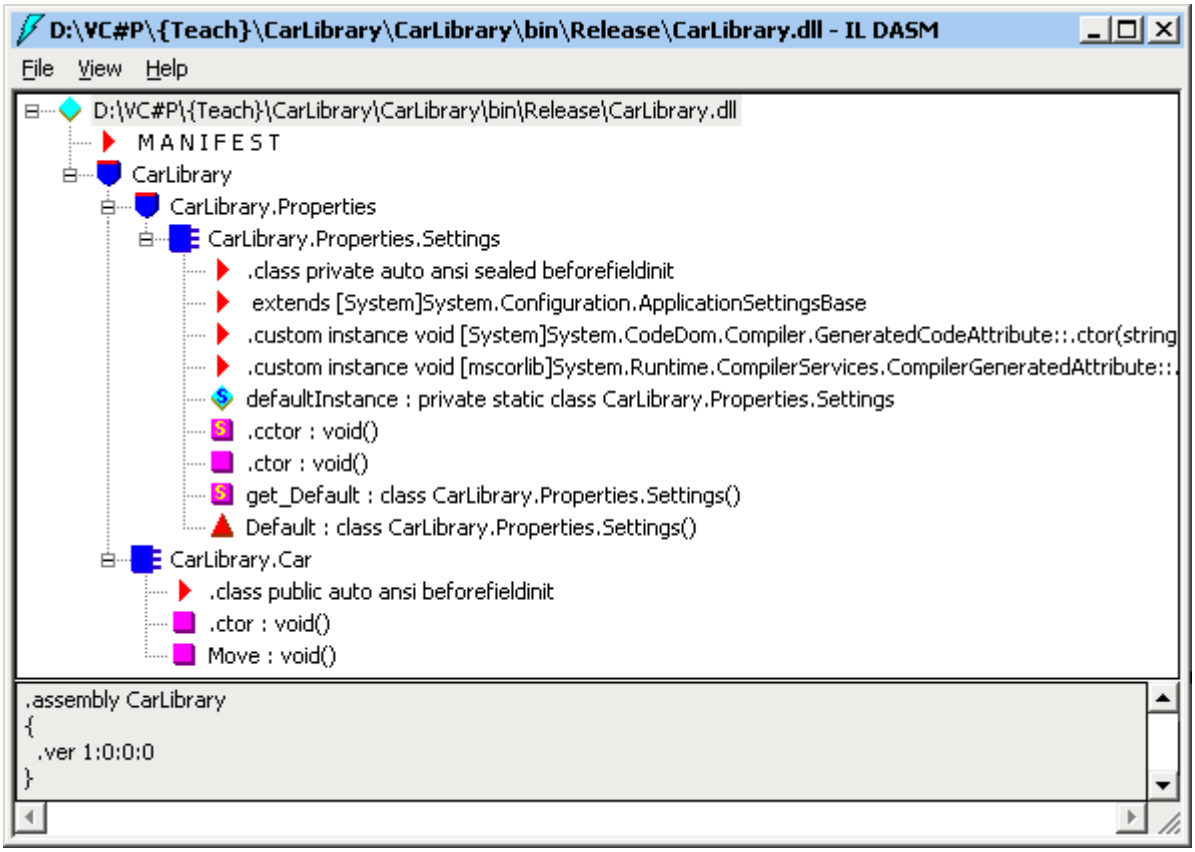

Рисунок 5 — Утилита ILDasm

Чтобы посмотреть манифест сборки, нужно дважды щелкнуть на слово MANIFEST.

Для описания манифеста используются следующие теги:

.assembly — объявление сборки.

.file — дополнительные теги этой сборки.

.class extern — классы, объявленные в этой сборке, но экспортируемые другой сборкой.

.exeloc — местонахождение исполняемого файла сборки.

.mainfestres — внутренние ресурсы сборки.

.module — объявление модуля (без манифеста).

.module extern — внешние модули (модули данной сборки, на которые есть ссылки внутри текущего модуля).

.assembly extern — внешние сборки (необходимые для нормальной работы данной сборки).

.publickey — содержит открытый ключ.

.publickeytoken — содержит маркер открытого ключа.

Чтобы посмотреть инструкции IL метода move, например, также нужно дважды щелкнуть на метод:

```
.method public hidebysig instance void  Move() cil managed
{
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  ldstr      "Hello\n"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
} // end of method Car::Move
```

Чтобы выгрузить иерархию типов в текстовый файл, нужно выбрать в меню программы ILDasm File — Dump TreeView. При этом вместо значков дерева иерархии будут использованы следующие текстовые обозначения:

 — (.dot)

 — (NSP) namespace

 — (CLS) class

 — (VCL) value class (struct)

 — (INT) interface

 — (MET) method

 — (STM) static method

 — (PTY) property

 — (FLD) field

 — (STF) static field

Чтобы выгрузить в текстовый файл типы вместе с инструкциями IL, нужно выбрать в меню программы ILDasm File — Dump.

Чтобы просмотреть метаданные типов, нужно в программе ILDasm ввести сочетание клавиш Ctrl+M.

1.6 Пространства имен

Пространство имен (namespace) объявляет некоторую область видимости. Пространства имен могут быть вложенными.

На языке C# пространство имен задается конструкцией:

```
namespace name[.name1] ...] { type-declarations }
```

Внутри пространства имен могут быть определены:

- другие пространства имен;
- классы;
- интерфейсы;
- структуры;
- перечисления;
- делегаты.

Наиболее важные пространства имен библиотеки типов:

System — низкоуровневые классы для работы с простыми типами, выполнения математических операций, сборки мусора и т.п.

System.Collections — работа с контейнерами.

System.Data — для работы с базами данных.

System.Drawing — для работы с графикой (GDI+).

System.IO — для выполнения операций ввода-вывода.

System.Net — для передачи данных по сети.

System.Reflection — для обнаружения, создания и вызова пользовательских типов во время выполнения.

System.Runtime — для взаимодействия с традиционным кодом.

System.Runtime.Remoting — для удаленного доступа.

System.Security — средства безопасности.

System.Threading — для работы с потоками.

System.Web — для работы с Web.

System.Windows.Forms — для работы с окнами.

System.XML — для работы с XML.

Пространство имен в C# включается инструкцией ***using***, например:

```
using System.Threading
```

или квалифицированным указанием типа, например:

```
System.Threading.Thread.CurrentThread
```

2 Элементы языка C#

2.1 Типы

Типы делятся на структурные (value types), ссылочные (reference types) и указательные (pointer types). Структурные типы языка приведены в таблице 1.

Таблица 1 — Структурные типы языка C#

Псевдоним C#	Соответствие CLS	Системный тип	Примечание
sbyte	–	SByte	8 бит, беззнаковое
byte	+	Byte	8 бит, знаковое
short	+	Int16	16 бит, знаковое
ushort	–	UInt16	16 бит, беззнаковое
int	+	Int32	32 бит, знаковое
uint	–	UInt32	32 бит, беззнаковое
long	+	Int64	64 бит, знаковое
ulong	–	UInt64	64 бит, беззнаковое
char	+	Char	16 бит символ Unicode
float	+	Single	
double	+	Double	
bool	+	Boolean	true или false
decimal	+	Decimal	128 бит, знаковое
struct			упрощенный класс

Структурные типы могут также содержать значение ***null*** (указывающее на отсутствие значения). Типы, которые могут не иметь значения, являются производными от структуры ***System.Nullable***. Тип, который может иметь значение ***null***, обозначается при помощи знака ?:

```
static void Main() {
    int? num = null;
    if (num.HasValue == false)
        Console.WriteLine("num = Null");
    int x = num.GetValueOrDefault();
    try {
        x = num.Value;
    }
    catch (InvalidOperationException e) {
        Console.WriteLine(e.Message);
    }
}
```

Ссылочные типы приведены в таблице 2.

Таблица 2 — Ссылочные типы языка C#

Ключевое слово	*Соответствие CLS*	*Системный тип*	*Примечание*
class			
delegate			Указатель на функцию-делегат
interface			Абстрактный класс, содержащий только сигнатуры методов, индексаторов и событий
object	+	Object	Любой тип есть объект
string	+	String	Содержит символы Unicode

Указательные типы могут быть использованы только внутри небезопасного (***unsafe***) контекста. Можно использовать указатели на любой тип из таблицы 1, кроме struct, на тип перечисления, на указательный тип, на структуру, которая содержит только типы таблицы 1, кроме ***struct***. Указательные типы не происходят от ***System.Object***, и нет соответствующей операции преобразования. Объявление указательных типов отличается от принятого в языке C:

```
int* p1, p2, p3;
```

Указательный тип не может указывать на ссылочный тип или на структуру, которая содержит ссылочный тип.

Небезопасный контекст образуется внутри метода (включая параметры) при помощи модификатора ***unsafe*** (модификаторы см. ниже), или объявлением небезопасного блока:

```
unsafe {
    /* небезопасный контекст */
}
```

Тип void указывает на отсутствие возвращаемого значения и не может быть использован в качестве параметра метода. Используется также внутри небезопасного контекста.

2.2 Модификаторы

Модификаторы доступа:

private — доступ только из класса-контейнера;

public — нет ограничений доступа;

protected — доступ только из класса-контейнера и производных;

internal — доступ в пределах текущей сборки;

protected internal — сочетание protected и internal.

В таблице 1 приведены разрешенные модификаторы доступа.

Таблица 1 — Разрешенные модификаторы доступа

Элемент	*Модификатор доступа по умолчанию*	*Разрешено использовать модификатор доступа*
enum	public	никакой
interface	public	никакой
struct	private	public, internal, private
class	private	любой

Другие модификаторы см. таблицу 2.

Таблица 2 — Модификаторы

Модификатор	*Применяется к*	*Значение*
abstract	класс, метод, свойство, индексатор, событие	Реализуется в производном классе.
const	поле, переменная	Неизменяемое значение.
event	делегат	Вызываемая функция.
extern	метод	Реализован вне сборки.
new	метод	Отменяет полиморфное поведение.
override	метод, свойство, индексатор, событие	Замещение элемента базового класса. Элемент должен быть virtual, abstract или override.
readonly	поле	Только для чтения.
sealed	класс, метод, свойство	Завершает иерархию. Метод или свойство должны иметь модификатор override и замещать элемент базового класса.
static	класс, поле, метод, свойство, оператор, событие, конструктор	Статический элемент принадлежит типу в целом.
unsafe	контекст	Используется операция с указателями.
virtual	метод, свойство, индексатор, событие	Виртуальный элемент, который может быть замещен в производном классе.
volatile	поле	Может изменяться множеством конкурирующих потоков. Не подвергается оптимизации.

В таблице 2 поле обозначает элемент данных.

2.3 Операторы

Оператор if имеет отличия от аналогичного оператора C++. В качестве логического значения можно использовать только логические типы и значения. Например, следующий оператор будет неправильным:

```
if (1) {}
```

Правильным будет следующий оператор:

```
if (true) {}
```

или оператор

```
if (1 == 1) {}
```

Оператор for аналогичен оператору C++, за исключением того, что параметр цикла, объявленный в операторе, не виден за пределами цикла:

```
for (int i = 0; i < 5; i++) {
}
//i = 1; // недопустимо
```

Новый оператор цикла foreach/in предназначен для просмотра элементов массива или коллекции:

```
int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };
foreach (int i in fibarray) {
    System.Console.WriteLine(i);
}
```

Оператор ***switch***, ***do***, ***while***, ***break***, ***continue***, ***goto*** и ***return*** аналогичны соответствующим операторам C++.

Операторы управления исключениями ***throw***, ***try*** и ***catch*** аналогичны соответствующим операторам языка C++, за исключением того, что выбрасываемые исключения должны быть производными от ***System.Exception***. Новый оператор ***finally*** описывает блок, который выполняется в любом случае после блока ***try***, независимо от того, было выброшено исключение в блоке ***try***, или нет.

Новым является контекст, в котором проверяется наличие или отсутствие переполнения арифметической операции. Для этой цели используются ключевые слова ***checked*** и ***unchecked***. Переполнение фиксируется при выполнении операций ++, —, –, +, *, /, унарный минус и явное (***explicit***) приведение типа.

В следующем примере используется контекст ***checked***, в котором генерируется исключение в случае переполнения:

```
try {
    z = checked(x + y);
}
```

```
        catch (System.OverflowException e) {
            Console.WriteLine(e.ToString());
        }
```

2.4 Ключевые слова аргументов

Аргументы методов могут иметь модификаторы ***ref***, ***out*** и ***params***.

Модификатор ***ref*** обозначает параметр, передаваемый по ссылке.

Модификатор ***out*** обозначает параметр, передаваемый по ссылке, который инициализируется в методе. Если метод имеет ***out*** или ***ref*** параметр, то при вызове метода нужно указывать модификатор.

Модификатор ***params*** указывает на параметр, который принимает множество фактических значений. При этом метод не может иметь никаких других параметров. Например:

```
    public static void UseParams(params int[] list) {
        for (int i = 0 ; i < list.Length; i++) {
            Console.WriteLine(list[i]);
        }
        Console.WriteLine();
    }
    static void Main() {
        UseParams(1, 2, 3);
    }
```

1.5 Ключевые слова операций

Модификатор ***explicit*** описывает операцию явного приведения одного типа в другой. Например:

```
public static explicit operator Celsius(Farenheit f) {
    return new Celsius((5.0f/9.0f)*(f.degrees-32));
}
```

Данное приведение используется следующим образом:

```
Farenheit f = new Farenheit(100.0f);
Celsius c = (Celsius)f;
```

Модификатор ***implicit*** описывает операцию неявного приведения одного типа в другой. Например:

```
class MyType {
    public static implicit operator int(MyType m) {
        // преобразование из MyType в int
    }
}
```

Данное приведение используется следующим образом:

```
MyType x;
int i = x;
```

Модификатор ***operator*** используется для перегрузки операций в структурах и классах. Используется в одной из следующих форм:

```
public static result-type operator unary-operator ( op-type operand )
public static result-type operator binary-operator (
      op-type operand,
      op-type2 operand2
)
public static implicit operator conv-type-out ( conv-type-in operand )
public static explicit operator conv-type-out ( conv-type-in operand )
```

Например, для класса дробных чисел Fraction:

```
class Fraction {
    int num, den;
    public Fraction(int num, int den) {
        this.num = num;
        this.den = den;
    }
    public static Fraction operator +(Fraction a, Fraction b) {
        return new Fraction(a.num * b.den + b.num * a.den,
           a.den * b.den);
    }
    public static Fraction operator *(Fraction a, Fraction b) {
        return new Fraction(a.num * b.num, a.den * b.den);
    }
    public static implicit operator double(Fraction f) {
        return (double)f.num / f.den;
    }
}
```

2.6 Ключевые слова классов

Модификатор ***base*** определяет элемент базового класса (например, конструктор):

```
public class B {
    public B() { }
    public B(int) { }
    public int GetNum() { }
}
public class D : B {
    public D() : base() { }
    public D(int i) : base(i) { base.GetNum(); }
}
```

Модификатор ***this*** указывает на текущего представителя класса.

2.7 Ключевые слова разных контекстов

get указывает на функцию, которая возвращает значение свойства.

Например:

```
get { return some_value; }
```

set указывает на функцию, которая устанавливает значение свойства.

value указывает на новое значение свойства.

Например:

```
set { some_var = value; }
```

partial указывает на часть определения класса, структуры или интерфейса. Другая часть при этом определяется в других файлах.

Например:

```
public partial class Employee {
    public void DoWork() { }
}
public partial class Employee {
    public void GoToLunch() { }
}
```

where определяет ограничение типа, который требуется в данном контексте.

Например, требует реализации определенного интерфейса:

```
public class MyGenericClass<T> where T:IComparable { }
```

или использования определенных базовых классов:

```
class MyClassy<T, U>
    where T : class
    where U : struct
{
}
```

3 Классы C#

2.8 Структуры

Структуру в C# можно условно рассматривать как упрощенный класс. Структуры могут содержать поля (элементы данных), константы, методы, свойства, индексаторы, операторы, события, конструкторы и вложенные типы.

Структура не может иметь конструктора по умолчанию и деструктора, не может наследовать структуру или класс и не может выступать в качестве базового класса, но может реализовывать интерфейсы. Структуры не могут иметь модификатор ***protected***. Структуры могут быть созданы без ключевого слова new. Элементы структуры нельзя инициализировать при описании.

При передаче структуры как параметра создается копия, а не ссылка.

2.9 Классы

Класс — это описание пользовательского типа, состоящее из методов, свойств, полей, событий, делегатов и вложенных классов.

Для описания класса используется один или несколько модификаторов и ключевое слово ***class***.

Классы могут наследовать только один базовый класс, но множество интерфейсов. Описание класса может быть распределено между несколькими модулями (файлами).

Классы могут содержать *вложенные* классы и структуры. Вложенные классы имеют доступ к закрытым и защищенным элементам контейнерного класса через представителя контейнерного класса. Передать этого представителя можно через конструктор вложенного класса. Например:

```
public class Parent {
    private int m_int = 0;
    public Parent() { }
    public Parent(int m) { m_int = m; }
    public class Inner {
        private Parent parent;
        public Inner() { }
        public Inner(Parent parent) { this.parent = parent; }
        public int iprop {
            get { return parent.m_int; }
            set { parent.m_int = value; }
        }
    }
}
```

Полное имя вложенного класса ***Parent.Inner***. Пример использования:

```
        Parent p = new Parent();
```

```
Parent.Inner pi = new Parent.Inner(p);
pi.iprop = 9;
Console.WriteLine(pi.iprop.ToString());
```

Статические классы содержат только статические элементы и не могут быть базовыми. Они используются для описания функциональности, не требующей создания объектов (например, класс математических методов). Статический класс может содержать только статические элементы.

```
static class CompanyInfo {
    public static string CompanyName() { return "CompanyName"; }
    public static string CompanyAddress() { return "CompanyAddress"; }
}
```

Конструкторы класса могут быть открытыми, закрытыми и статическими. Класс может определять сколько угодно конструкторов с параметрами и один конструктор без параметров (конструктор по умолчанию).

Закрытый конструктор класса препятствует созданию представителя класса. Статический конструктор используется для инициализации статических элементов класса или для выполнения однократных действий. Статический конструктор вызывается автоматически перед созданием первого представителя класса или обращением к статическому элементу класса.

Статический конструктор не может иметь модификаторов или параметров, его нельзя вызвать.

В C# нет *конструктора копии*. Вместо этого нужно создать метод для копирования объекта. Однако класс может иметь конструктор с параметром-ссылкой на класс:

```
public class Person {
    private string name;
    public Person(string name) { this.name = name; }
    public Person(Person another) { name = another.name; }
    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Используется этот конструктор для создания копии существующего объекта при инициализации:

```
Person p = new Person("Ann");
Person p2 = new Person(p);
p.Name = "Mary";
Console.WriteLine(p.Name + "-" + p2.Name);
```

Свойства классов имитируют доступ к полям класса, но являются специальными методами, называемыми ***accessor***. Пример см. описанный выше класс Person. Методы одного свойства могут иметь разные модификаторы доступа (асимметричные свойства). Например:

```
    public string Name {
        get { return name; }
        protected set { name = value; }
    }
```

При этом доступ к методу свойства должен быть не выше, чем у самого свойства. Если реализуется наследуемое от интерфейса свойство, модификаторы свойства запрещены, однако, если интерфейс определяет только один метод свойства, второй метод может иметь модификатор, например:

```
interface IPerson {
    string Name {
        get;
    }
}
public class Person : IPerson {
    private string name;
    public string Name {
        get { return name; }
        protected set { name = value; }
    }
}
```

Если свойство наследуется не от абстрактного свойства, то доступ к нему может быть ограничен, при этом свойство производного класса скрывается, например:

```
public class Person {
    private string name = "noname";
    public string Name {
        get { return name; }
        set { name = value; }
    }
}
public class Manager : Person {
    private string name;
    public Manager(string n) { name = n; }
    private string Name {
        get { return name; }
        set { name = value; }
    }
}
class Program {
    static void Main() {
        Manager m = new Manager("Ann");
        Console.WriteLine(m.Name);
    }
}
```

Классы могут также определять *константы*, в качестве которых можно использовать структурные типы, тип перечисления или ссылки ***null***.

```
    public const int months = 12;
    public enum days { Monday, Tuesday };
    public const days monday = days.Monday;
    public const Person nullperson = null;
```

Константы должны быть инициализированы, при этом можно использовать значения других констант без образования циклических ссылок.

Классы, объявленные с модификатором ***abstract***, являются *абстрактными*. Представителя абстрактного класса создать нельзя. Назначение абстрактного класса — служить в качестве базового.

Абстрактный класс может определять *абстрактные методы и свойства*, которые должны быть определены в производном классе при помощи модификатора ***override*** или ***new***. Пример:

```
public abstract class Person {
    protected string name = "noname";
    public abstract void SetName(string n);
    public abstract string Name {
        get; set;
    }
}
public class Manager : Person {
    public Manager(string n) { name = n; }
    public override void SetName(string n) { name = n; }
    public override string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Если абстрактный класс является производным от класса, определяющего виртуальный метод, он может заместить виртуальный метод абстрактным, например:

```
public class BaseClass {
    public virtual void Draw() { /* original implementation */ }
}
public abstract class AbstractClass : BaseClass {
    public abstract override void Draw();
}
public class DerivedClass : AbstractClass {
    public override void Draw() { /* new implementation */ }
}
```

При наследовании абстрактного виртуального метода его виртуальность сохраняется в производных классах.

Если класс наследует интерфейс, использование модификатора ***override*** запрещено, а использование модификатора ***new*** разрешено, например:

```
interface point {
    int X { get; set; }
    int Y { get; set; }
}
public class Point : point {
    private int x, y;
    public int X {
        get { return x; }
        set { x = value; }
```

```
    }
    public new int Y {
        get { return y; }
        set { y = value; }
    }
}
```

Элементы класса, объявленные с модификатором ***new***, *закрывают* собой элементы базового класса. Например:

```
public class Shape {
    public static int id = 1;
    public virtual void Draw() { Console.WriteLine("Shape"); }
}
public class Circle : Shape {
    new public static int id = 2;
    public override void Draw() { Console.WriteLine("Circle"); }
}
public class Square : Shape {
    new public static int id = 3;
    new public void Draw() { Console.WriteLine("Square"); }
}

class Program {
    static void Main() {
        Circle c = new Circle();
        Square s = new Square();
        c.Draw(); // Circle
        s.Draw(); // Square
        Shape[] sh = new Shape[2];
        sh[0] = new Circle();
        sh[1] = new Square();
        foreach (Shape o in sh) {
            o.Draw(); // Circle - Shape
        }
    }
}
```

Классы, объявленные с модификатором ***sealed***, не могут служить в качестве базовых или абстрактных. Методы или свойства, объявленные с модификатором ***sealed***, отменяют их виртуальность для производных классов. При этом подразумевается, что метод или свойство замещают элементы базового класса.

2.10 Индексаторы

Индексаторы позволяют индексировать элементы классов и структур способом, используемым для массивов. Индексатор похож на свойство, но методы индексатора имеют параметры. Например:

```
public class MyArray<T> {
    private T[] ar = new T[10];
    public T this[int i] {
        get {
```

```
            if (i >= 0 && i < 11) return ar[i]; else return ar[0];
        }
        set {
            if (i >= 0 && i < 11) ar[i] = value;
        }
    }
}
class Program {
    static void Main() {
        MyArray<int> a = new MyArray<int>();
        a[0] = 9;
        int b = a[0];
    }
}
```

Индексаторы могут иметь произвольное количество произвольных параметров. Индексаторы могут быть перегружаемыми. Пример объявления индексатора в интерфейсе:

```
interface Indexer {
    int this[int index] { get; set; }
}
public class IndexerClass : Indexer {
    private int[] ar = new int[10];
    public int this[int index] {
        get { return ar[index]; }
        set { ar[index] = value; }
    }
}
```

2.11 Делегаты

Делегат — это тип, ссылающийся на метод (указатель на функцию). На тип делегата указывает его имя. Делегату можно назначить (присвоить) любой метод, который соответствует типу делегата (его сигнатуре).

Делегаты являются типами, производными от класса ***MulticastDelegate*** (многозначный делегат). Некоторые наследуемые делегатом элементы:

Method — свойство, возвращает имя метода, на который указывает делегат.

Target — если делегат указывает на элемент класса, то возвращает имя класса. Если делегат указывает на статический метод, возвращает ***null***.

GetInvocationList — возвращает массив типов ***Delegate***, каждый из которых представляет запись во внутреннем списке указателей на функции делегата.

Combine — статический метод для создания делегата, указывающего на несколько разных методов.

Remove — статический метод для удаления делегата из списка указателей на функции.

Делегаты используются в случаях:

1) Генерирование событий (см. ниже).

2) Когда желательна инкапсуляция статического метода.

3) Когда нет необходимости использовать элементы класса кроме тех, на которые ссылается делегат.

4) Требуется несколько реализаций метода класса.

5) В целях упрощения кода класса.

Сущность делегатов поясняет следующий пример.

```
public class Point {
    public int x, y;
    public void SetValues(int X, int Y) { x = X; y = Y; }
}
public class Point2 {
    public int x, y;
    public void SetValues(int X, int Y) { x = X; y = Y; }
}
class Program {
    public delegate void Dlg(int a, int b);
    static void Main() {
        Point p1 = new Point();
        Dlg d = p1.SetValues;
        d(1, 1);
        Point2 p2 = new Point2();
        d += p2.SetValues;
        d(2, 2);
        Point p3 = new Point();
        d += p3.SetValues;
        d(3, 3);
        d -= p.SetValues;
        d(4, 4);
    }
}
```

К представителям делегатов операции "+" и "–" применимы как к обычным переменным, что поясняет следующий вариант метода *Main*:

```
    static void Main() {
        Point p1 = new Point();
        Point2 p2 = new Point2();
        Point p3 = new Point();
        Dlg d1 = p1.SetValues;
        Dlg d2 = p2.SetValues;
        Dlg d3 = p3.SetValues;
        Dlg d = d1 + d2 + d3;
        d(5, 5);
    }
```

Делегаты могут использовать *анонимные* методы (в версии C# не ниже 2.0). Анонимные методы записываются в блок { }, непосредственно следующий за инициализацией делегата, например:

```
delegate void Printer(string s);
class TestClass {
    static void Print(string s) { Console.WriteLine(s); }
```

```
        static void Main() {
            Printer p = delegate(string s) { Console.WriteLine(s); };
            p("Вызывает анонимный метод.");
            p = new Printer(TestClass.Print);
            p("Вызывает именованный метод.");
        }
    }
```

Типичный пример — создание потока, код которого описывается анонимно, например:

```
using System;
using System.Threading;
namespace threads {
    class TestClass {
        static void Main() {
            Thread t1 = new Thread( delegate() {
                Console.Write("Hello, ");
                Console.WriteLine("World!");
            } );
            t1.Start();
        }
    }
}
```

Рассмотрим пример класса ***Point***, операции с которым выполняются при помощи делегатов типа ***PointOperation***:

```
    public delegate void PointOperation(Point a, Point b);
    public class Point {
        public int x, y;
        public Point(int a, int b) { x = a; y = b; }
        public override string ToString() { return x + "," + y; }
        public void Operate(PointOperation od, Point b) {
            od(this, b);
        }
    }
```

Операции описываются с сигнатурой делегата ***PointOperation*** как статические методы класса программы. Следующая функция описывает сложение двух объектов типа ***Point***:

```
        static void PointAdd(Point a, Point b) {
            a.x += b.x;
            a.y += b.y;
        }
```

В методе ***Main*** создается делегат ***add***, который может быть использован двояким способом — самостоятельно, и как параметр метода ***Operate***:

```
            Point p1 = new Point(1, 2);
            Point p2 = new Point(3, 4);
            PointOperation add = PointAdd;
            add(p1, p2);
            Console.WriteLine(p1);
            p1.Operate(add, p2);
```

```
        Console.WriteLine(p1);
```

Учитывая возможность самостоятельного использования делегата в данном случае, класс ***Point*** может и не содержать метод ***Operate***. Однако в случае, если используется класс, описывающий множество объектов, например, типа ***Point***, такой метод может оказаться полезным.

```
public class Points {
    private Point[] ps;
    public Points(int n) {
        ps = new Point[n];
        for (int i = 0; i < n; i++) ps[i] = new Point(0, 0);
    }
    public void Operate(PointOperation od, Point b) {
        foreach (Point a in ps) {
            od(a, b);
        }
    }
}
```

Для выполнения операций используется делегат ***add*** и новый делегат ***prn***, который указывает на следующий статический метод:

```
    static void PointPrint(Point a, Point b) {
        Console.WriteLine(a);
    }
```

В методе ***Main*** создается новый делегат и используется для вывода значений:

```
        Points ps = new Points(2);
        ps.Operate(add, p2);
        PointOperation prn = PointPrint;
        ps.Operate(prn, p2);
```

Использование делегата в этом случае может быть еще прозрачнее, если определить делегат для операции с одним объектом ***Point***, например:

```
public delegate void PointOp(Point a);
```

При этом в классе ***Points*** определяется новый метод:

```
    public void Operate2(PointOp od) {
        foreach (Point a in ps) {
            od(a);
        }
    }
```

Делегат для выполнения операции может быть описан как метод другого класса, например, класса аффинных преобразований:

```
public class Trans {
    private int x, y;
    public Trans(int a, int b) { x = a; y = b; }
    public void Transform(Point a) {
        // аффинные преобразования
```

```
            a.x += x; a.y -= y;
        }
    }
```

В методе ***Main*** создается представитель аффинных преобразований ***t***, делегат ***po*** и передается представителю класса ***Points***:

```
            Trans t = new Trans(1, 1);
            PointOp po = t.Transform;
            ps.Operate2(po);
            ps.Operate(prn, p2);
```

1.12 События

Событие — это способ уведомления об изменениях, происходящих внутри представителей классов. События используют делегатов для обеспечения безопасной в отношении типов инкапсуляции методов, которые будут вызваны.

В следующем примере класс ***Point*** определяет событие ***Change***, имеющее тип делегата ***ChangeEvent***.

```
public delegate void ChangeEvent();
public class Point {
    public event ChangeEvent Change;
    private int x, y;
    public void SetValues(int a, int b) {
        x = a;
        y = b;
        if (Change != null) Change();
    }
}
```

Событие вызывается при изменении состояния объекта в ***SetValues***.

Программа определяет метод ***ChangeHandler***, подходящий под описание делегата ***ChangeEvent***. Этот метод назначается делегату ***Change***, и используется для отслеживания изменений в объектах ***Point***.

```
class Program {
    static void ChangeHandler() { Console.WriteLine("Changed"); }
    static void Main() {
        Point p = new Point();
        p.Change += new ChangeEvent(ChangeHandler);
        p.SetValues(1, 1);
    }
}
```

Делегат события может иметь параметры и возвращаемое значение, как и любой делегат. Обычная практика использовать в качестве параметров события объект-источник и некоторые аргументы типа ***EventArgs***:

```
public delegate void ChangeEvent(object sender, EventArgs e);
public class Point {
```

```
    public event ChangeEvent Change;
    public int x, y;
    public void SetValues(int a, int b) {
        x = a;
        y = b;
        Change(this, EventArgs.Empty);
    }
}
public class PointsManager {
    public void ReceiveChangeEvent(object sender, EventArgs e) {
        Console.WriteLine(sender.ToString());
    }
}
class Program {
    static void ChangeHandler(object s, EventArgs e) {
        Console.WriteLine(((Point)s).x + "," + ((Point)s).y);
    }
    static void Main() {
        Point p = new Point();
        PointsManager pm = new PointsManager();
        p.Change += new ChangeEvent(ChangeHandler);
        p.Change += pm.ReceiveChangeEvent;
        p.SetValues(1, 1);
    }
}
```

Использование стандартных методов является обязательным для взаимодействия в среде .NET. События, использующие стандартный соглашения, могут использовать также стандартный делегат события ***EventHandler***. Пусть, например, есть следующий класс ***Point2***:

```
public class Point2 {
    public event EventHandler Changed;
    public int x, y;
    public Point2(int a, int b) { x = a; y = b; }
    public override string ToString() { return x + "," + y; }
    public void SetValues(int a, int b) {
        x = a; y = b;
        EventHandler temp = Changed;
        if (temp != null) {
            temp(this, EventArgs.Empty);
        }
    }
}
```

Он определяет событие типа ***EventHandler***, которое удовлетворяет стандартным соглашениям. Перед генерацией события нужно создать временный объект типа ***EventHandler***, и присвоить ему значение делегата события. Значение временной переменной должно проверяться на значение ***null***. События обрабатываются любым методом, который удовлетворяет стандартным соглашениям, например:

```
    static void Handler1(object sender, EventArgs e) {
        Console.WriteLine(sender);
    }
```

Этот метод далее можно присоединить к событию объекта стандартным способом:

```
Point2 p = new Point2(0, 0);
p.Changed += new EventHandler(Handler1);
p.SetValues(3, 7);
```

События на самом деле подобны свойствам — они состоят из двух методов типа *accessor*: *add* для добавления делегата и *remove* для удаления. В некоторых случаях эти методы требуют переопределения.

1.13 Обобщенные классы

Обобщенный класс (*generic class*) — это параметризованный относительно типа шаблон класса. В отличие от шаблонов C++ обобщенные классы C# имеют много ограничений и обладают меньшей мощностью, зато большей безопасностью в отношении типов.

В библиотеке базовых классов обобщенные классы широко применяются в коллекциях пространства имен ***System.Collections.Generic***.

Описание обобщенного класса имеет вид:

```
class A<T> [where T : ограничение] { . . . }
```

Параметры обобщенных классов могут иметь ограничения (*restraints*):

1) ограничение на ссылочный тип разрешает использовать в качестве параметра только ссылочные типы:

```
class A<T> where T : class { . . . }
```

2) Ограничение на структурный тип разрешает использовать в качестве параметра только структурный тип (который не является *nullable*):

```
class A<T> where T : struct { . . . }
```

3) ограничение на конструктор по умолчанию разрешает использовать в качестве параметра только тип, имеющий конструктор по умолчанию:

```
class A<T> where T : new() { . . . }
```

4) ограничение на базовый класс разрешает использовать в качестве параметра только тип, который является производным от заданного класса.

```
class A<T> where T : <тип> [, <тип> ] { . . . }
```

5) ограничение на интерфейс разрешает использовать в качестве параметра только тип, который наследует заданный интерфейс.

```
class A<T> where T : <интерфейс> [, <интерфейс> ] { . . . }
```

6) открытое ограничение (*naked constraint*) задает параметр обобщенного типа, который должен быть типом другого параметра:

```
class A<T, U> where T : U { . . . }
```

Обобщенный тип может иметь одновременно несколько ограничений.

Обобщенными могут быть не только классы, но и методы. Например, следующий обобщенный метод позволяет работать с элементами разных структурных типов:

```
static void Swap<T>(ref T a, ref T b) where T : struct {
    T c = a; a = b; b = c;
}
```

Следующий обобщенный метод позволяет использовать различные коллекции, наследующие ***IList<>***, для какой-то обработки:

```
static void ProcessList<T>(IList<T> coll) {
    foreach (T item in coll) {
        /* какая-то обработка */
    }
}
```

Обобщенные классы являются строго типизированными, что ведет к определенным ограничениям, но проявляется в свойстве таких классов быть безопасными относительно типов (*type safety*).

Например, при помощи базового класса ***ArrayList*** можно создать коллекцию объектов произвольного типа, что далее может привести к их несовместимости:

```
ArrayList list = new ArrayList(); // Неправильно!!!
list1.Add(5);
list1.Add("str");
int sum = 0;
foreach (int x in list) {
    sum += x;
}
```

При помощи обобщенного базового класса такая ситуация невозможна, поскольку тип элементов коллекции указывается во время объявления объекта и контролируется во время компиляции:

```
List<int> list = new List<int>(); // Правильно!!!
list1.Add(5);
// нельзя добавить строку list1.Add("str");
```

В некоторых случаях строгая типизация приводит к невозможности выполнить некоторые действия, которые на самом деле выполнить можно.

Например, если есть список типа ***List<object>***, то его нельзя объединить со списком типа ***List<int>***. С точки зрения типов это возможно, поскольку ***List<int>*** является производным от типа ***IEnumtrable<int>***, который, в свою очередь, является производным от типа ***IEnumtrable<object>***.

В подобных случаях рекомендуется использовать какой-нибудь вспомогательный обобщенный относительно конфликтующих типов метод, при помощи которого можно выполнить преобразование объектов к одному типу, после чего операция станет возможной. В описываемом случае можно, например, определить следующий метод для слияния списков:

```
static void Add<S, T>(List<S> source, List<T> target) where S : T {
    foreach (S element in source) {
        target.Add(element);
    }
}
```

Делегаты также могут быть типизированными, например:

```
public delegate void DelegateType<T>(T item);
public static void Notify(int item) { . . . }
DelegateType<int> d1  = new DelegateType<int>(Notify);
DelegateType<int> d2  = Notify; // упрощенный синтаксис
```

Поскольку типы обобщенных классов могут быть произвольными, в случае инициализации нулевыми значениями требуется различать ссылочные и структурные типы, так как они имеют разные нулевые значения. Для этой цели используется ключевое слово ***default***, с помощью которого элемент данных получает правильное значение по умолчанию:

```
private T = default(T);
```

В обобщенных классах нельзя:

• использовать явные значения типов в качестве параметров;

• определять арифметические операции;

• использовать операции, которые могут привести к конфликту типов; это возможно, только если тип параметра эти операции подразумевает;

• использовать в качестве типов параметров обобщенные типы;

• использовать параметр в качестве базового типа;

• использовать параметр определенного типа.

1.14 Незначащие типы

Структурные типы, которые в дополнение к типовым значениям могут принимать значение null, называются здесь *незначащими*. Незначащие типы являются производными от структуры ***System.Nullable***. Для обозначения незначащих типов используется знак ***?***. Например:

```
private int? num = null;
private int y = 0;
```

Чтобы определить, какое значение имеет незначащий тип, используется два способа. Первый заключается в обращении к свойству ***HasValue*** (есть значение), например:

```
if (num.HasValue) y = num.Value; else y = -1;
```

Другой способ заключается в попытке получить значение и в случае ошибки поймать исключение ***InvalidOperationException***:

```
try {
    y = num.Value;
} catch (InvalidOperationException) {
    y = -1;
}
```

Свойство ***GetValueOrDefault*** возвращает значение типа, если оно есть, или значение по умолчанию для данного типа, если значение равно ***null***, например:

```
private int? num = null;
private int y = num.GetValueOrDefault; // y == default(int);
```

Специальный оператор ***??*** возвращает заданное в операторе значение по умолчанию в случае, если незначащий тип присваивается значащему и первый имеет незначащее значение ***null***, например:

```
int? a = null;
int? b = 22;
int y = a ?? -1;  //// y == -1
a = 33;
y = a ?? -1;      //// y == 33
a = null;
y = a ?? b ?? -1; //// y == 22
```

Если незначащим типом является тип ***bool***, то его нельзя использовать в качестве условия:

```
bool? a = 1;
// запрещено // if (a) { . . . }
```

Однако его можно использовать в выражении:

```
bool? a = 1;
if (a == null) { . . . }
```

Результат выполнения операций ***&*** и ***|*** с типом ***bool?*** приведен в следующей таблице:

x	*y*	*x & y*	*x | y*
true	*null*	*null*	*true*
null	*true*	*null*	*true*
false	*null*	*false*	*null*
null	*false*	*false*	*null*
null	*null*	*null*	*null*

4 Потоки

1.15 Домены

Домены — это средство изоляции приложений и их ресурсов друг от друга. Один процесс может состоять из нескольких доменов приложений, которые будут также изолированы друг от друга, как отдельные процессы. Ближайшими аналогами доменов являются *апартаменты* COM. Домены в значительной степени способствуют расширяемости (*scalability*) серверных программных компонентов.

Обычно домены создаются и управляются компонентами системы времени исполнения. Программно домены реализуются при помощи типа *System.AppDomain*. В следующем примере создается домен приложения, в который загружается предварительно скомпилированная сборка ***C:\HW.exe*** и запускается на исполнение:

```
static void Main() {
    // Создаем домен
    AppDomain domain = AppDomain.CreateDomain("DomainHW");
    // загружаем и исполняем
    domain.ExecuteAssembly(@"C:\HW.exe");
    // выгружаем
    AppDomain.Unload(domain);
}
```

Правила относительно доменов:

• Сборка перед исполнением должна быть загружена в домен; это сделает либо среда исполнения CLR, либо приложение;

• Ошибка в приложении одного домена не влияет на работу других доменов процесса;

• Отдельные приложения могут быть остановлены и выгружены без остановки процесса;

• Нельзя выгрузить отдельные сборки или типы, выгрузить можно только домен целиком;

• Код одного домена не имеет доступа к коду и другим ресурсам другого домена. Объекты, которые передаются между доменами, либо копируются, либо заменяются прокси-объектами. В первом случае обращение к объекту является локальным относительно домена, во втором — удаленным. При этом используется средства межпроцессного взаимодействия. Доступом к объектам других доменов управляет класс *MarshalByRefObject*;

• Атрибуты безопасности, назначенную коду приложения, равно как и поведение кода, контролируются доменом, в котором код исполняется;

• Очевидно, что не существует ресурсов, принадлежащих процессу, так как они принадлежат домену, а точнее — приложениям домена.

Важнейшие элементы класса *AppDomain*:

CreateDomain — создает домен в текущем процессе.

CreateInstance — создает представителя указанного типа указанной сборки.

ExecuteAssembly — запускает указанную сборку на выполнение.

Load — загружает сборку в домен приложения.

Unload — выгружает домен приложения.

CurrentDomain — возвращает домен текущего потока.

GetCurrentThreadID — возвращает идентификатор текущего потока.

Чтобы выполнить код другого приложения, нужно либо загрузить сборку в текущий домен, либо создать новый домен и загрузить сбоку в него. Например, есть следующая сборка:

```
namespace HWRemote {
    public class RemoteObject : System.MarshalByRefObject {
        public RemoteObject() {
            System.Console.WriteLine("HW - RemoteObject ctor");
        }
    }
    class Program {
        static void Main() {
            System.Console.WriteLine("HW - Main");
        }
    }
}
```

Запустить сборку в домене приложения можно следующим образом:

```
using System;
using System.Reflection;
namespace HWRemoteTest {
    class Program {
        static void Main() {
            // загружаем сборку в текущий домен
            Assembly a = Assembly.LoadFrom(@"C:\HWRemote.exe");
            // создаем представителя HelloWorldRemote
            a.CreateInstance("HWRemote.RemoteObject");
        }
    }
}
```

Запустить сборку в новом домене можно следующим образом:

```
namespace HWRemoteTest2 {
    class Program {
        static void Main() {
            // создаем домен
            AppDomain domain = AppDomain.CreateDomain("HWDomain");
            // загружает сборку и исполненяет Main
            domain.ExecuteAssembly(@"D:\HWRemote.exe");
            // создает представителя RemoteObject
            domain.CreateInstanceFrom(@"D:\HWRemote.exe",
                "HWRemote.RemoteObject");
        }
```

```
    }
}
```

Сборку также можно подключить при помощи диалога *Project — Add Reference*. В ***using*** нужно добавить ссылку на пространство имен добавленного проекта. Далее создать представителя сборки можно при помощи генератора ***new***:

```
namespace HWAddReference {
    using HWRemote;
    class Program {
        static void Main() {
            HWRemote.RemoteObject o = new RemoteObject();
        }
    }
}
```

Домены и потоки

Домен формирует границы, внутри которых исполняется защищенный, безопасный и надежный управляемый код. Поток — это структура операционной системы, которая используется CLR для выполнения кода.

Между доменом и потоком нет однозначного соответствия. Несколько потоков могут исполняться внутри одного домена. Конкретный поток не ограничивается рамками одного домена. Иначе говоря, *потоки могут свободно пересекать границы доменов*. Для доменов не создаются новые собственные потоки.

Поток всегда исполняется в домене. Система времени исполнения отслеживает, в каком домене какие потоки исполняются.

Атрибуты

Атрибут — это декларативная информация, приписываемая к классу, методу, свойству и т.п. Атрибуты могут быть прочитаны во время исполнения при помощи рефлексии типов. Есть два вида атрибутов: атрибуты, определенные CLR и пользовательские атрибуты. Атрибуты заключаются в квадратные скобки, например:

```
[System.Serializable]
public class SampleClass { /* описание сериализуемого объекта */ }
```

Атрибуты добавляются к метаданным приложения. Обычно используются для взаимодействия с COM и другим унаследованным кодом.

Рефлексия типов

Рефлексия типов — это процесс, с помощью которого во время исполнения можно найти тип и информацию о нем. С помощью рефлексии

типов можно загрузить сборку и получить информацию о ней. Пусть есть сборка типа ***dll***:

```
using System;
public class MyAssembly {
    public void MyMethod() {
        Console.WriteLine("This is MyMethod");
    }
}
```

Следующая программа перечисляет методы сборки:

```
using System;
using System.Reflection;
using System.Security.Permissions;
namespace EgAssemblyTest {
    class Program {
        [PermissionSetAttribute(SecurityAction.Demand, Name =
"FullTrust")]
        public static void Main() {
            Assembly a = Assembly.Load("MyAssembly");
            Type[] mytypes = a.GetTypes();
            BindingFlags flags = (BindingFlags.Public |
                BindingFlags.Instance);
            foreach (Type t in mytypes) {
                MethodInfo[] mi = t.GetMethods(flags);
                Object obj = Activator.CreateInstance(t);
                foreach (MethodInfo m in mi) {
                    Console.WriteLine(m);
            }
        }
    }
}
```

1.16 Потоки

Управление потоками позволяет параллельно выполнять множество конкурирующих потоков. Для управления потоками используются классы и интерфейсы пространства имен *System.Threading*. С их помощью можно создавать и запускать потоки, синхронизировать, приостанавливать и завершать.

Для создания потока используется конструктор, который указывает на подходящий метод, например:

```
Thread t = Thread(some_object.some_method);
```

Метод ***Start*** запускает поток на выполнение:

```
t.Start();
```

Пример класса, который формирует поток и контролирует его выполнение:

```
public class Worker {
```

```
    // синхронизированная переменная-признак работы потока
    private volatile bool running = false;
    // метод, используемый в качестве функции потока
    public void process() {
        Console.WriteLine("now enter...");
        running = true;
        while (running) {
            Console.WriteLine("...processing...");
        }
        Console.WriteLine("now leave...");
    }
    // метод, сигнализирующий о необходимости завершить поток
    public void stop_running() {
        running = false;
    }
}
```

Для создания потока используется, например, статический метод ***Main***:

```
Worker w = new Worker();
Thread t = new Thread(w.process);
t.Start();
// ждем активизации
while (!t.IsAlive);
// даем время поработать, пока основной поток спит
Thread.Sleep(1);
// запрашиваем завершение потока
w.stop_running();
// ждем завершения потока, приостанавливая основной
t.Join();
Console.WriteLine("main thread ends");
```

Для создания потоков могут быть использованы также два делегата. Делегат ***ThreadStart*** представляет поток без параметров. Делегат ***ParameterizedThreadStart*** позволяет запускать потоки с параметрами.

Пример использования делегата ***ThreadStart***:

```
class ServerClass {
    public void InstanceMethod() {
        Console.WriteLine("ServerClass.InstanceMethod is running.");
        Thread.Sleep(2000);
        Console.WriteLine("The instance method has ended.");
    }
    public static void StaticMethod() {
        Console.WriteLine("ServerClass.StaticMethod is running.");
        Thread.Sleep(3000);
        Console.WriteLine("The static method has ended.");
    }
}
public class Simple {
    public static void Main() {
        Console.WriteLine("Thread Simple.");
        ServerClass so = new ServerClass();
        Thread ic = new Thread(new ThreadStart(so.InstanceMethod));
        ic.Start();
        Console.WriteLine("Instance method started.");
```

```
        Thread sc = new Thread(new
ThreadStart(ServerClass.StaticMethod));
        sc.Start();
        Console.WriteLine("Static method started.");
    }
}
```

Использовать делегат ***ParameterizedThreadStart*** не рекомендуется по причинам возможного нарушения безопасности типов. Вместо этого рекомендуется создать оберточный класс (*wrap class*, *helper class*), который сформирует параметры метода, например:

```
public class ThreadWithState {
    // информация, необходимая потоку
    private string boilerplate;
    private int value;
    // конструктор получает необходимую информацию
    public ThreadWithState(string text, int number) {
        boilerplate = text;
        value = number;
    }
    public void ThreadProc() {
        Console.WriteLine(boilerplate, value);
    }
}
public class Program {
    public static void Main() {
        // конструктор получает параметры
        ThreadWithState tws = new ThreadWithState("Число = {0}.", 42);
        Thread t = new Thread(new ThreadStart(tws.ThreadProc));
        t.Start();
        Console.WriteLine("Основной поток выполняет работу и ждет.");
        t.Join();
        Console.WriteLine("Метод завершен, основной поток
завершается.");
    }
}
```

Важнейшие элементы класса ***Thread***:

CurrentThread — возвращает ссылку на текущий поток;

GetData(), ***SetData()*** — возвращает (записывает) значения в заданный слот;

GetDomain() — возвращает ссылку на домен приложения;

GetDomainId() — возвращает идентификатор домена приложения;

Sleep() — приостанавливает выполнение текущего потока;

IsAlive — возвращает признак запуска потока;

IsBackground — возвращает признак фонового потока;

Name — дружественное имя потока;

Priority — управляет приоритетом потока;

ThreadState — состояние потока из перечисления ***ThreadState***;

Join() — блокирует поток до завершения другого потока;

Resume() — возобновляет выполнение потока;

Suspend() — приостанавливает выполнение потока.

Следующий пример показывает использование слотов:

```
using System;
using System.Threading;
class Slot {
    static Random randomGenerator;
    static LocalDataStoreSlot localSlot;
    static Slot() {
        randomGenerator = new Random();
        localSlot = Thread.AllocateDataSlot();
    }
    public static void SlotTest() {
        // записывает разные данные в слоты потоков
        Thread.SetData(localSlot, randomGenerator.Next(1, 200));
        // выводит данные слота потока
        Console.WriteLine("Данные слота {1,3} потока {0}",
            AppDomain.GetCurrentThreadId().ToString(),
            Thread.GetData(localSlot).ToString());
        // пусть другие потоки запишут данные в слот
        Thread.Sleep(1000);
        // выводит данные слота потока
        Console.WriteLine("Данные слота {1,3} потока {0}",
            AppDomain.GetCurrentThreadId().ToString(),
            Thread.GetData(localSlot).ToString());
    }
}
class Program {
    static void Main() {
        Thread[] ts = new Thread[4];
        for (int i = 0; i < ts.Length; i++) {
            ts[i] = new Thread(new ThreadStart(Slot.SlotTest));
            ts[i].Start();
        }
    }
}
```

Синхронизация потоков

Синхронизация — это приостановка выполнения потока до того момента, когда освободится критическая секция некоторых данных. Критическая секция в данном контексте — это часть кода потока, которая обращается к данным, являющихся общими для нескольких конкурирующих параллельных потоков (разделяемыми).

Для безопасного доступа к общим данным используются синхронизирующие средства операционных систем, такие, как критические секции, семафоры, мьютексы, сигналы, функции *WaitXXX* и т.п. Среда .NET Framework версии 2.0 предоставляет несколько средств для синхронизации потоков, реализующих системные средства синхронизации в классах пространства имен ***System.Threading***.

Класс Interlocked

Выполняет атомарные операции над числами (целыми и вещественными. Основные методы:

Add — складывает два числа и заменяет первое число суммой;

Decrement — уменьшает число на единицу;

Exchange — заменяет значение числа новым;

Increment — увеличивает число на единицу.

Есть еще несколько других операций.

Ключевое слово lock

Заблокировать часть кода метода от прерывания другими потоками можно при помощи ключевого слова ***lock***. Для этой цели требуется вспомогательный объект (в примере ***lock_object***), который блокируется на все время исполнения кода следующего за ним блока:

```
public void Method() {
    object lock_object = new Object();
    lock(lock_object) { /* монопольный доступ к объекту */ }
}
```

Блокируемый объект определяет область действия блокировки. В примере эта область распространяется на метод, поскольку область действия объекта ограничивается методом.

Варианты применения блокировки следующие:

1) ***lock(some_object)*** — самый безопасный метод

2) ***lock(this)*** — небезопасно для public объектов

3) ***lock(typeof(some_type))*** — небезопасно для public элементов

4) ***lock("some_string")*** — небезопасно

Блокировка соответствует примитиву ***CRITICAL_SECTION*** Win32 API.

Объект блокировки на самом деле уникальным образом идентифицирует разделяемый множеством потоков ресурс. На практике объект блокировки часто представляет ресурс, требующий синхронизации потоков. Например, если некоторый объект (представитель класса) является объектом синхронизации, его можно использовать качестве объекта блокировки.

В качестве объектов блокировки не рекомендуется использовать открытые (***public***) типы и объекты, или объекты, управление которыми лежит за пределами приложения. Например, нежелательно использовать блокировку ***lock(this)***, т.к. это может привести к блокировке объекта извне приложения, если объект является открытым (public).

Не рекомендуется блокировать строковые литералы, поскольку интерпретатор CLR создает один экземпляр строкового литерала для всех

одинаковых литералов. Блокировка любого из них блокирует одновременно все другие экземпляры во всем приложении.

Объект блокировки лучше всего выбирать из закрытых и защищенных элементов классов и их представителей. Классы могут иметь для этой цели специальные элементы. Например, класс ***Array***, а также многие стандартные коллекции обладают свойством ***SyncRoot***, которое возвращает предназначенный для блокировки объект:

```
Array a = new int[] { 1, 2, 4 };
lock(a.SyncRoot) {
    /* монопольный доступ */ foreach (Object o in a) { }
}
```

Мониторы

Ключевое слово ***lock*** на самом деле является реализацией монитора. Метод ***Enter*** монитора обеспечивает защиту последующего кода от прерывания другими потоками, метод ***Exit*** выводит поток из критической секции. При входе в критическую секцию используются блоки ***try-finally***:

```
Object obj = (Object)x;
Monitor.Enter(obj);
try {
    // монопольный доступ
} finally {
    Monitor.Exit(obj);
}
```

В данной реализации монитора он может оказаться в состоянии бесконечного ожидания освобождения критической секции. При помощи метода ***TryEnter*** можно задать время ожидания входа в критическую секцию, по истечении которого попытка входа отменяется, например:

```
if (Monitor.TryEnter(this, 300)) {
    try {
        // монопольный доступ
    } finally {
        Monitor.Exit(this);
    }
} else {
    // не удалось войти в монитор
}
```

Монитор сразу входит в критическую секцию, если в качестве объекта синхронизации используется, например, целочисленная переменная (в примере переменная ***x***):

```
private int x;
Monitor.Enter(x);
try {
} finally {
```

```
}
```

Для каждого блокируемого объекта монитор отслеживает:

• очередь потоков, готовых захватить объект (готовые потоки);

• очередь потоков, ожидающих уведомления об изменении блокировки объекта (ожидающие потоки).

Метод ***Wait*** монитора позволяет разблокировать критическую секцию и захватить объект потоком из очереди готовых потоков, при этом текущий поток блокируется до наступления некоторого события.

Методы ***Pulse*** и ***PulseAll*** сигнализируют ожидающим потокам об изменении состояния блокируемого объекта.

Если необходимо заблокировать весь код метода, методу можно задать атрибут *Synchronized*, при этом отпадает необходимость в использовании методов *Enter* и *Exit*:

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void Method() { /* монопольный доступ */ }
```

Синхронизирующие события

Объекты, которые могут находиться в одном из двух состояний, — сигнальном и несигнальном, называются событиями. События используют для приостановки и возобновления исполнения потоков. Поток приостанавливается (блокируется), если он ожидает перехода некоторого события в сигнальное состояние.

Есть два типа событий — сбрасываемые автоматически (***AutoResetEvent***) и вручную (***ManualResetEvent***). Автоматически сбрасываемое событие переходит в несигнальное состояние после того, как ожидающий его поток возобновит свое исполнение. События с ручным сбросом позволяют возобновить исполнение множества потоков, после чего они могут быть переведены в несигнальное состояние при помощи метода ***Reset*** (в сигнальное состояние событие переводится при помощи метода ***Set***).

Поток переводится в состояние ожидания при помощи одного из следующих методов ***System.Threading.WaitHandle***:

1) ***WaitOne*** — поток блокируется до тех пор, пока одно событие не перейдет в сигнальное состояние;

2) ***WaitAny*** — поток блокируется до тех пор, пока одно из событий не перейдет в сигнальное состояние;

3) ***WaitAll*** — поток блокируется до тех пор, пока все события не перейдут в сигнальное состояние.

В следующем примере поток ожидает перевода события ***autoEvent*** с ручным сбросом в сигнальное состояние:

```
class Test {
```

```
    // синхронизирующее событие
    static AutoResetEvent autoEvent;
    static void process() {
        Console.WriteLine("поток ожидает события");
        autoEvent.WaitOne();
        Console.WriteLine("поток возобновлен и завершается");
    }
    static void Main() {
        autoEvent = new AutoResetEvent(false);
        Thread t = new Thread(process);
        t.Start();
        Thread.Sleep(1000);
        Console.WriteLine("основной поток сигнализирует");
        autoEvent.Set();
    }
}
```

Мьютексы

Мьютексы, в отличие от мониторов, используются для синхронизации доступа к ресурсам, разделяемым между процессами. Для синхронизации доступа к ресурсам процесса (домена, приложения) следует использовать мониторы — они, например, требуют меньше накладных расходов.

Мьютексы .NET Framework являются объектной оболочкой для мьютексов, создаваемых операционной системой. Для использования мьютекса за границами процесса он должен быть поименованным, например:

```
public static Mutex mutex = new Mutex(true, "my_mutex");
```

Первый параметр типа ***bool*** обозначает, что мьютекс захвачен текущим потоком. Пример использования мьютекса:

```
        private static Mutex mut = new Mutex();
        private const int numIterations = 1;
        private const int numThreads = 3;
        private static void MyThreadProc() {
            for (int i = 0; i < numIterations; i++) {
                UseResource();
            }
        }
        private static void UseResource() {
            mut.WaitOne();
            Console.WriteLine("{0} enter", Thread.CurrentThread.Name);
            Thread.Sleep(500);
            Console.WriteLine("{0} leave", Thread.CurrentThread.Name);
            mut.ReleaseMutex();
        }
        static void Main() {
            for (int i = 0; i < numThreads; i++) {
                Thread myThread = new Thread(new
ThreadStart(MyThreadProc));
                myThread.Name = String.Format("П{0}", i + 1);
                myThread.Start();
            }
        }
```

Использование пула потоков

```
public class Fibonacci {
    public int N { get { return _n; } }
    private int _n;
    public int FibOfN { get { return _fibOfN; } }
    private int _fibOfN;
    private ManualResetEvent _doneEvent;
    public Fibonacci(int n, ManualResetEvent doneEvent) {
        _n = n; _doneEvent = doneEvent;
    }
    public void ThreadPoolCallback(Object threadContext) {
        int threadIndex = (int)threadContext;
        Console.WriteLine("Протоκ {0} start", threadIndex);
        _fibOfN = Calculate(_n);
        Console.WriteLine("Поток {0} ready", threadIndex);
        _doneEvent.Set();
    }
    public int Calculate(int n) {
        if (n <= 1) return n;
        return Calculate(n - 1) + Calculate(n - 2);
    }
}
public class ThreadPoolExample {
    static void Main() {
        const int fcount = 10;
        ManualResetEvent[] doneEvents = new ManualResetEvent[fcount];
        Fibonacci[] fibArray = new Fibonacci[fcount];
        Random r = new Random();
        Console.WriteLine("Задач - {0}.", fcount);
        for (int i = 0; i < fcount; i++) {
            doneEvents[i] = new ManualResetEvent(false);
            Fibonacci f = new Fibonacci(r.Next(20, 40), doneEvents[i]);
            fibArray[i] = f;
            ThreadPool.QueueUserWorkItem(f.ThreadPoolCallback, i);
        }
        WaitHandle.WaitAll(doneEvents);
        Console.WriteLine("Вычисслены все числа.");
        for (int i = 0; i < fcount; i++) {
            Fibonacci f = fibArray[i];
            Console.WriteLine("Fibonacci({0}) = {1}", f.N, f.FibOfN);
        }
    }
}
```