Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

БАЗОВЫЕ ТЕХНОЛОГИИ ПЛАТФОРМЫ .NET

Курс лекций
для студентов специальности
1-40 01 03 Информатика и технологии программирования

Минск 2013

Содержание

1. Работа с числами

Платформа .NET содержит базовый набор типов для представления чисел. В пространстве имён `System` определены целочисленные типы `SByte`, `Int16`, `Int32`, `Int64`, `Byte`, `UInt16`, `UInt32`, `UInt64`, типы для чисел с плавающей запятой `Single` и `Double`, тип повышенной точности `Decimal`. Для каждого из типов язык C# предлагает псевдоним в виде ключевого слова (например, `int` для `Int32`).

Целочисленные типы поддерживают арифметические и битовые операции, операции сравнения и методы преобразования (в частности, из строки в число). В каждом типе определены константы `MinValue` и `MaxValue`.

```
uint x;
if (uint.TryParse(Console.ReadLine(), out x))
{
    if (x > byte.MaxValue) x &= 255;
}
```

Внутренний формат типов `float` и `double` отвечает стандарту IEEE 754. Эти типы содержат константы `MinValue`, `MaxValue`, `Epsilon` (наименьшее положительное число), `NaN` («не число»), `NegativeInfinity`, `PositiveInfinity`.

```
// вычислим сумму геометрической прогрессии со знаменателем 0.1
double sum = 0.0, xn = 1.0;
while (xn > double.Epsilon)
{
    sum += xn;
    xn *= 0.1;
}
```

Тип `decimal` использует 96 бит для хранения основания, 1 бит – для знака, 8 бит хранят позицию запятой в основании справа (число от 0 до 28). Этот тип содержит константы `MinValue`, `MaxValue`, `MinusOne`, `Zero`, `One`. В Common Intermediate Language нет элементарных инструкций для манипулирования типом `decimal`, операции с ним транслируются в вызовы методов.

Пространство имён `System.Numerics` содержит структуры `BigInteger` и `Complex`. Структура `BigInteger` определяет целое число неограниченной длины. Экземпляр структуры может быть создан на основе строки, массива байтов, или путём приведения одного из обычных целых типов. Структура `BigInteger` выполняет перегрузку арифметических и битовых операций и содержит несколько статических методов, соответствующих математическим функциям (например, `Sign()`, `Abs()`, `DivRem()`, `Pow()`, `Log()`).

```
// подсчёт факториала 10000
BigInteger factorial = 1;
for (int i = 2; i <= 10000; i++)
{
    factorial *= i;
}
```

Структура `Complex` служит для представления комплексного числа и обладает набором стандартных элементов (перегрузка арифметических операций, некоторые математические функции, свойства для действительной и мнимой части, модуля и аргумента).

```
Complex z1 = new Complex(3, 5);
Complex z2 = new Complex(-2, 10);
Complex z3 = Complex.Sin(z1/z2);     // синус от частного двух чисел
Console.WriteLine(z3.Magnitude);     // напечатаем модуль числа
```

В статическом классе `System.Math` содержится набор методов, соответствующих основным математическим функциям (табл. 1).

Таблица 1

Элементы класса `System.Math`

Имя элемента	Описание
`Abs()`	Модуль (функция перегружена для аргумента `sbyte`, `short`, `int`, `long`, `float`, `double`, `decimal`)
`Acos()`, `Asin()`, `Atan()`	Арккосинус, арксинус, арктангенс в радианах для аргумента `double`. Если указан недопустимый аргумент, возвращается `double.NaN`
`Atan2()`	Арктангенс, вычисленный по отношению двух аргументов
`BigMul()`	Произведение двух аргументов типа `int`, имеющее тип `long`
`Ceiling()`	Наименьшее целое, которое больше или равно указанному аргументу (функция перегружена для аргумента `double` и `decimal`)
`Cos()`, `Sin()`, `Tan()`	Косинус, синус, тангенс
`Cosh()`, `Sinh()`, `Tanh()`	Гиперболические косинус, синус и тангенс
`DivRem()`	Вычисляет частное и остаток при делении двух чисел типа `int` или `long`
`E`	Константа *e*
`Exp()`	Экспонента
`Floor()`	Наибольшее целое, которое меньше или равно указанному аргументу (функция перегружена для аргумента `double` и `decimal`)
`IEEERemainder()`	Остаток от деления, вычисленный по правилам стандарта IEEE 754
`Log()`	Логарифм, вычисленный по заданному основанию (или натуральный логарифм, если указан один аргумент)
`Log10()`	Десятичный логарифм
`Max()`, `Min()`	Наибольшее и наименьшее из двух чисел (функция перегружена для всех числовых типов, кроме `char`)
`PI`	Константа π
`Pow()`	Возводит число в указанную степень
`Round()`	Округление до ближайшего целого. Можно задать дополнительный аргумент, определяющий поведение в случае, если аргумент лежит ровно посредине между двумя целыми числами
`Sign()`	Знак числа (-1, 0 или 1) (функция перегружена для аргумента `sbyte`, `short`, `int`, `long`, `float`, `double`, `decimal`)
`Sqrt()`	Квадратный корень
`Truncate()`	Округление до ближайшего целого числа в направлении нуля

Класс `System.Random` генерирует псевдослучайную последовательность значений `byte`, `int` или `double`. Конструктор класса `Random` перегружен и может принимать целочисленное начальное значение (*зерно*) для инициализации последовательности псевдослучайных чисел. Применение одинакового зерна гарантирует генерирование одной и той же последовательности, что иногда необходимо в целях отладки. Если зерно явно не указано, используется значение, вычисленное по текущему времени. Метод `Next()` генерирует случайное целое число, при этом можно задать допустимый интервал. Метод `NextDouble()` возвращает случайное вещественное число из интервала [0, 1), а метод `NextBytes()` заполняет массив байтов случайными значениями.

```
Random r = new Random(1000);
int x = r.Next() + r.Next(100) + r.Next(-10, 10);
double y = r.NextDouble();
byte[] buffer = new byte[10];
r.NextBytes(buffer);
```

Отметим, что в задачах криптографии следует использовать более сильный генератор случайных чисел, чем `Random`. Например, в пространстве имён `System.Security.Cryptography` имеется генератор `RandomNumberGenerator`:

```
var rand = RandomNumberGenerator.Create();
byte[] bytes = new byte[32];
rand.GetBytes(bytes);        // заполняем массив случайными байтами
```

2. Представление даты и времени

В пространстве имён `System` определены три неизменяемые структуры для работы с датой и временем: `TimeSpan`, `DateTime` и `DateTimeOffset`.

Структура `TimeSpan` хранит интервал времени в виде отсчётов по 100 наносекунд. Для создания экземпляра структуры можно использовать один из её конструкторов, статические методы вида `TimeSpan.From`*`ЕдиницыВремени`*`()` или разность двух переменных типа `DateTime`.

```
var ts1 = new TimeSpan(50);          // количество отсчётов по 100 нс
var ts2 = new TimeSpan(1, 20, 50);   // часы, минуты, секунды
var ts3 = TimeSpan.FromHours(3.5);
```

Структура `TimeSpan` перегружает операции сравнения и аддитивные операции. Свойства структуры `TimeSpan` позволяют обратиться либо к отдельному временному компоненту интервала (свойства `Days`, `Hours`, `Minutes`, `Seconds`, `Milliseconds`), либо выразить весь интервал через указанную единицу времени (`TotalDays`, `TotalHours` и т.п.).

```
TimeSpan ts = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);
Console.WriteLine(ts.Days);              // 9
Console.WriteLine(ts.Seconds);           // 59
Console.WriteLine(ts.TotalDays);         // 9.99998842592593
```

Структуры DateTime и DateTimeOffset предназначены для хранения даты и времени. Интервал дат – от 1 января 1 года до 31 декабря 9999 года, точность времени – 100 наносекунд. DateTimeOffset дополнительно хранит *смещение всемирного координированного времени* (UTC offset), что даёт возможность правильно сравнивать даты разных часовых поясов.

Структура DateTime имеет конструкторы, позволяющие указать для создаваемого экземпляра год, месяц, день, часы, минуты, секунды, миллисекунды. Дополнительно можно задать аргумент из перечисления DateTimeKind, который показывает, какое именно время содержит экземпляр (Unspecified, Local, Utc). Также можно задать используемый датой календарь – это влияет на алгоритм вычисления некоторых свойств даты. Конструкторы структуры DateTimeOffset дополнительно разрешают указать смещение UTC как значение TimeSpan (целое количество минут)[1].

```
var dt1 = new DateTime(2012, 12, 21);
var dt2 = new DateTime(2012, 12, 21, 12, 0, 0, DateTimeKind.Local);
var dt3 = new DateTime(2012, 12, 21, new PersianCalendar());
var dto = new DateTimeOffset(dt1, TimeSpan.FromHours(-6));
```

Определено неявное преобразование DateTime к DateTimeOffset. Такие свойства структуры DateTimeOffset как UtcDateTime, LocalDateTime, DateTime дают возможность получить значение времени UTC в виде DateTime.

Используя структуры DateTime и DateTimeOffset, текущее время и дату можно узнать при помощи свойств Now, Today, UtcNow. Обе структуры содержат свойства для чтения отдельных временных компонентов, а также методы для увеличения временных компонент на указанную (возможно, отрицательную) величину. Структуры DateTime и DateTimeOffset также перегружают операции сравнения и аддитивные операции.

```
var dt = DateTime.Today;                // текущая дата (без времени)
Console.WriteLine(dt.Month);            // узнаем текущий месяц
Console.WriteLine(dt.DayOfWeek);        // и день недели
var changed_dt = dt.AddDays(-3.5);
DateTimeOffset dto = DateTime.Now;      // неявное преобразование
Console.WriteLine(dto.Offset);          // узнаем UTC offset
```

3. Работа со строками и текстом

Для представления отдельных символов в платформе .NET применяется структура System.Char, которая использует Unicode-кодировку UTF-16. Язык C# предлагает для типа Char псевдоним char.

В структуре Char имеется экземплярный метод сравнения CompareTo(). Большинство статических методов структуры Char нужны для выяснения при-

[1] Классы TimeZone и TimeZoneInfo предоставляют информацию, касающуюся названий часовых поясов, смещений UTC и правил перехода на летнее время.

надлежности символа к одной из Unicode-категорий (таблица символов и категорий доступна по адресу blackbeltcoder.com/Resources/CharClass.aspx). Многие методы перегружены: они могут принимать в качестве аргумента либо отдельный символ, либо строку и номер символа в ней.

Таблица 2

Статические методы структуры `System.Char`

Имя метода	Описание
`ConvertFromUtf32()`	Преобразует целочисленный суррогатный UTF-код в строку
`ConvertToUtf32()`	Преобразует пару суррогатных символов[1] в UTF-код
`GetNumericValue()`	Возвращает численное значение символа, если он является цифрой, и –1.0 в противном случае
`GetUnicodeCategory()`	Метод возвращает элементы перечисления `UnicodeCategory`, описывающего категорию символа
`IsControl()`	Возвращает `true`, если символ является управляющим
`IsDigit()`	Возвращает `true`, если символ является десятичной цифрой
`IsHighSurrogate()`	Определяет, является ли символ старшим символом-заместителем суррогатной пары
`IsLetter()`	Возвращает `true`, если символ является буквой
`IsLetterOrDigit()`	Возвращает `true`, если символ является буквой или цифрой
`IsLower()`	Возвращает `true`, если символ – это буква в нижнем регистре
`IsLowSurrogate()`	Определяет, является ли символ младшим символом-заместителем суррогатной пары
`IsNumber()`	Возвращает `true`, если символ является десятичной или шестнадцатеричной цифрой
`IsPunctuation()`	Возвращает `true`, если символ является знаком препинания
`IsSeparator()`	Возвращает `true`, если символ является разделителем
`IsSurrogate()`	Возвращает `true`, если символ является суррогатным
`IsSurrogatePair()`	Определяет, образуют ли два заданных символа суррогатную пару
`IsSymbol()`	Показывает, относится ли символ к категории символьных знаков
`IsUpper()`	Возвращает `true`, если символ – это буква в верхнем регистре
`IsWhiteSpace()`	Возвращает `true`, если символ является пробельным (например, пробел, символ конца строки, символ перевода каретки)
`Parse()`	Преобразует строку в символ. Строка должна состоять из одного символа, иначе генерируется исключение
`ToLower()`	Приводит символ к нижнему регистру
`ToLowerInvariant()`	Приводит символ к нижнему регистру, используя правила инвариантной культуры
`ToUpper()`	Приводит символ к верхнему регистру
`ToUpperInvariant()`	Приводит символ к верхнему регистру, используя правила инвариантной культуры
`TryParse()`	Пытается преобразовать строку в символ

[1] Некоторые символы Unicode представлены двумя 16-битными суррогатными символами.

Основным типом платформы .NET для работы со строками является класс `System.String` (псевдоним в C# – `string`). Этот класс представляет неизменяемый объект. Инициализация строки обычно выполняется при помощи строкового литерала. Однако класс `String` содержит конструкторы для создания строки из массива символов или символа, повторенного заданное число раз.

```
string literal = "This is a simple string";
string charArray = new string(new[] {'C', 'h', 'a', 'r'});
string charRepeated = new string('X', 3);
```

Для строк определены операции `==` и `!=`. При этом выполняется посимвольное сравнение строк – используется семантика типов значений.

```
string charArray = new string(new[] {'C', 'h', 'a', 'r'});
string charString = "Char";
Console.WriteLine(charArray == charString);    // True
```

Любая строка содержит индексатор для чтения отдельного символа. Кроме этого, символы строки можно перебрать при помощи цикла `foreach`.

```
foreach (char ch in "This is a simple string")
{
    Console.WriteLine(ch);
}
```

В языке C# соединение строк можно выполнить при помощи операции `+`, что транслируется в вызов метода `String.Concat()` (литералы соединяются на этапе компиляции).

Статические элементы класса `String` приведены в табл. 3.

Таблица 3

Статические элементы класса `System.String`

Имя элемента	Описание
`Empty`	Свойство только для чтения, которое возвращает пустую строку
`Compare()`	Сравнение двух строк. Реализации метода позволяют сравнивать как строки, так и подстроки. При этом можно учитывать регистр и алфавит конкретного языка
`CompareOrdinal()`	Сравнение двух строк. Реализации метода позволяют сравнивать как строки, так и подстроки. Сравниваются коды символов
`Concat()`	Конкатенация произвольного числа строк
`Concat<T>()`	Метод выполняет проход по указанной коллекции, преобразует каждый элемент в строку и соединяет полученные строки
`Copy()`	Создаёт копию строки
`Format()`	Выполняет форматирование строки в соответствии с заданными спецификациями формата
`Join()`	Соединение элементов массива строк или коллекции строк в единую строку. Между элементами можно вставить разделители

Окончание табл. 3

Имя элемента	Описание
`Join<T>()`	Метод выполняет проход по указанной коллекции, преобразует каждый элемент в строку и соединяет полученные строки. Между элементами можно вставить разделители
`IsNullOrEmpty()`	Проверяет, является ли аргумент пустой строкой или `null`
`IsNullOrWhiteSpace()`	Проверяет, является ли аргумент пустой строкой, `null` или состоит только из пробельных символов

Сводка экземплярных методов класса `String` приведена в табл. 4. Ни один из этих методов не меняет строку, у которой вызывается. Конечно, некоторые методы создают и возвращают в качестве результата новые строки.

Таблица 4

Экземплярные методы `System.String`

Имя метода	Описание
`CompareTo()`	Сравнивает строки для выяснения порядка
`Insert()`	Вставляет подстроку в заданную позицию
`Remove()`	Удаляет подстроку в заданной позиции
`Replace()`	Заменяет подстроку в заданной позиции на новую подстроку
`Split()`	Разбивает строку на массив слов. Допускает указание разделителя слов (по умолчанию – пробел), а также опции для удаления пустых слов из итогового массива
`Substring()`	Выделяет подстроку в заданной позиции
`CopyTo()`	Копирует указанный фрагмент строки в массив символов
`Contains()`	Определяет вхождение заданной подстроки
`IndexOf()`, `IndexOfAny()`, `LastIndexOf()`, `LastIndexOfAny()`	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
`StartsWith()`, `EndsWith()`	Возвращается `true` или `false`, в зависимости от того, начинается или заканчивается строка заданной подстрокой. При этом можно учитывать регистр и алфавит конкретного языка
`PadLeft()`, `PadRight()`	Выполняют «набивку» нужным числом пробелов в начале или в конце строки
`Trim()`, `TrimStart()`, `TrimEnd()`	Удаляются пробелы в начале и в конце строки или только с одного её конца
`ToCharArray()`	Преобразование строки в массив символов
`ToLower()`, `ToUpper()`, `ToLowerInvariant()`, `ToUpperInvariant()`	Изменение регистра символов строки

```
// один из способов перевернуть строку
string source = "This is a simple string";
char[] charArray = source.ToCharArray();
Array.Reverse(charArray);
string destination = new string(charArray);
```

Так как класс `String` представляет неизменяемый объект, многократное использование методов для модификации строки (например, в цикле) снижает производительность. В пространстве имён `System.Text` размещён класс `StringBuilder` для работы с редактируемой строкой. Класс `StringBuilder` применяет для хранения символов внутренний массив, который изменяется при редактировании строки (новые экземпляры объекта не создаются). Набор методов класса `StringBuilder` поддерживает основные операции редактирования строк (`Append()`, `Insert()`, `Remove()`, `Replace()`), а также получение «нормальной» строки из внутреннего массива.

```
// создаём объект на основе строки и указываем ёмкость массива
StringBuilder sb = new StringBuilder("start", 300);
for (int i = 1; i < 100; i++)
{
    sb.Append("abc");        // много раз добавляем к строке текст
}
string s = sb.ToString();    // получаем обычную строку
```

Платформа .NET предлагает поддержку работы с регулярными выражениями. *Регулярное выражение* (regular expression) – это строка-шаблон, которому может удовлетворять определённое множество слов. Регулярные выражения используются для проверки корректности информации (например, правильного формата адресов электронной почты), поиска и замены текста по определённому образцу.

Коротко опишем синтаксис построения регулярных выражений[1]. В простейшем случае регулярное выражение – это последовательность букв или цифр. Тогда оно определяет именно то, что представляет. Но, как правило, регулярное выражение содержит некоторые особые спецсимволы. Первый набор таких спецсимволов позволяет определить символьные множества (табл. 5).

Таблица 5

Символьные множества в регулярных выражениях

Выражение	**Значение**	**Обратное по смыслу выражение**
`[abcdef]`	Один символ из списка	`[^abcdef]`
`[a-f]`	Один символ из диапазона	`[^a-f]`
`\d`	Десятичная цифра (аналог `[0-9]`)	`\D`
`\w`	Словообразующий символ (зависит от текущей языковой культуры; например, для английского языка это `[a-zA-Z_0-9]`)	`\W`
`\s`	Пробельный символ (пробел, табуляция, новая строка, перевод каретки)	`\S`
`\p{…}`	Любой символ из указанной Unicode-категории. Например, `\p{P}` – символы пунктуации	`\P{…}`
`.`	Любой символ, кроме `\n`	`\n`

[1] Подробнее см. ru.wikipedia.org/wiki/Регулярные_выражения.

В регулярном выражении можно использовать особые символы для начала строки `^`, конца строки `$`, границы слова `\b`, символа табуляции `\t` и перевода строки `\n`.

Отдельные атомарные регулярные выражения допустимо обрамлять в *группы*, при помощи пары скобок `()`. Если необходимо, группы или выражения можно объединять, используя символ `|`.

К каждому символу или группе можно присоединить квантификаторы повторения:

`?`	повтор 0 или 1 раз;
`+`	повтор от 1 до бесконечности;
`*`	повтор от 0 до бесконечности;
`{n}`	повтор n раз ровно;
`{n, m}`	повтор от n до m раз;
`{n,}`	повтор от n раз до бесконечности;

Рассмотрим некоторые простые примеры регулярных выражений:

`^\s*$`	пустая строка
`\btext\b`	отдельное слово text
`\b[bcf]at\b`	слова bat, cat, fat

В .NET имеется пространство имён `System.Text.RegularExpressions`, содержащее набор типов для работы с регулярными выражениями. Основной тип для работы с регулярными выражениями – это класс `Regex`. Объект класса представляет одно регулярное выражение, которое указывается при вызове конструктора. Существует перегруженная версия конструктора, позволяющая указать различные опции для создаваемого регулярного выражения.

```
Regex re1 = new Regex(@"\b[bcf]at\b");
Regex re2 = new Regex(@"\b[bcf]at\b", RegexOptions.IgnoreCase);
```

Для поиска информации согласно текущему регулярному выражению можно использовать метод `Regex.IsMatch()`. Более продуктивным является применение функции `Match()`, которая возвращает объект класса `Match`.

```
Regex re = new Regex(@"\b[bcf]at\b", RegexOptions.Compiled);
Match match = re.Match("bad fat cat");
while (match.Success)
{
    Console.WriteLine(match.Index);
    Console.WriteLine(match.Value);
    match = match.NextMatch();
}
```

Для замены на основе регулярных выражений используется метод `Regex.Replace()`. Его аргументы – обрабатываемая строка и строка на замену.

4. Преобразование информации

Платформа .NET и языки программирования, построенные на её основе, предлагают богатый набор средств для решения задачи преобразования информации в данные различных типов.

Язык C# поддерживает операции явного и неявного преобразования типов, причём эти операции могут быть перегружены в пользовательских типах. При этом желательно, чтобы перегружаемые операции преобразования имели простую семантику и работали на сходных множествах значений.

Для выполнения взаимных преобразований данных *базовых типов* (числовые типы, `bool`, `string` и `DateTime`) предназначен статический класс `System.Convert`. Этот класс содержит набор методов вида `To`*`ИмяТипа`*`()`, где *`ИмяТипа`* является именем CLR для базовых типов. Каждый такой метод перегружен и принимает аргумент любого базового типа. Выполнение метода может быть успешным или генерировать исключения `InvalidCastException`, `FormatException`, `OverflowException`.

```
byte x = Convert.ToByte("123");          // x = 123
bool y = Convert.ToBoolean(10.5);        // y = true
int z = Convert.ToInt32(DateTime.Now);   // InvalidCastException
```

Класс `Convert` также содержит методы для взаимных преобразований массива байтов в строку (или массив символов) в формате Base-64:

```
// конвертируем массив в Base64 и выполним обратное преобразование
byte[] data = {10, 20, 30, 40, 50};
Console.WriteLine(Convert.ToBase64String(data));
data = Convert.FromBase64String("ChQeKDI=");
```

Отметим, что для унификации операций преобразования все базовые типы явным образом реализуют интерфейс `System.IConvertible`. Это интерфейс содержит набор методов вида `To`*`ИмяТипа`*`()`, где *`ИмяТипа`* – имя CLR для базовых типов. Каждый такой метод принимает аргумент типа `IFormatProvider`. Базовые типы при реализации `IConvertible` просто вызывают соответствующие методы класса `Convert`.

Статический класс `System.BitConverter` содержит набор методов для преобразования переменной числового или булевого типа в массив байтов, а также методы обратного преобразования. Подобные преобразования полезны при низкоуровневой работе с потоками данных.

```
byte[] data = BitConverter.GetBytes(10567);
int x = BitConverter.ToInt32(data, 0);
```

Важными видами преобразований являются получение строкового представления объекта и получение объекта из строки. Для получения строкового представления объекта можно использовать виртуальный метод `ToString()`, определённый в классе `System.Object`. Однако часто требуется дать возможность

выбора формата представления и учесть региональные стандарты. Для этой цели предназначен интерфейс `System.IFormattable`, который реализуют числовые типы, структура `DateTime`, класс `Enum` и некоторые другие типы.

```
public interface IFormattable
{
    string ToString(string format, IFormatProvider formatProvider);
}
```

Первый параметр метода `IFormattable.ToString()` – это строка, сообщающая способ форматирования объекта. Многие типы различают несколько строк форматирования. Например, структура `DateTime` поддерживает строку `"d"` для дат в кратком формате, `"D"` для дат в полном формате, `"T"` – для времени и т. д. Числа поддерживают строку `"X"` для шестнадцатеричного вывода, `"C"` – для валют, `"E"` – для научного формата. Числовые типы также различают шаблоны форматирования, чтобы отобразить нужное количество цифр и знаков в дробной части (полную информацию о строках форматирования смотрите в MSDN).

Второй параметр метода `IFormattable.ToString()` – это *поставщик формата*. Это объект, реализующий интерфейс `IFormatProvider`.

```
public interface IFormatProvider
{
    object GetFormat(Type formatType);
}
```

Поставщик формата знает, как обеспечить учёт региональных стандартов, влияющих на форматирование чисел, дат и времени. Метод `GetFormat()` возвращает нужный объект для форматирования по его типу. Например, объект класса `NumberFormatInfo` описывает группу свойств `NegativeSign`, `CurrencyDecimalSeparator`, `CurrencySymbol`, `NumberGroupSeparator` и т. п. У объекта класса `DateTimeFormatInfo` имеются свойства `Calendar`, `DayNames`, `DateSeparator`, `LongDatePattern`, `ShortTimePattern`, `TimeSeparator` и т. п.

Платформа .NET содержит несколько поставщиков формата, в частности, класс `CultureInfo`[1] . Конструктор этого класса принимает числовой или строковый идентификатор культуры. Свойство `CultureInfo.InvariantCulture` описывает инвариантную культуру, не связанную с региональными стандартами.

```
// дата, которую будем форматировать
var dt = new DateTime(2012, 12, 21);

// создаём поставщик формата для русской культуры
// список кодов: msdn.microsoft.com/en-us/goglobal/bb896001
var culture = new CultureInfo("ru-Ru");
```

[1] Классы `CultureInfo`, `NumberFormatInfo`, `DateTimeFormatInfo` определены в пространстве имён `System.Globalization`.

```
// s1 = "21 декабря 2012 г."
var s1 = dt.ToString("D", culture);

// используем инвариантную культуру
var s2 = dt.ToString("D", CultureInfo.InvariantCulture);

// используем культуру текущего потока выполнения
var s3 = dt.ToString("D", null);
```

Чаще всего при получении строкового представления объекта нужно указать только формат, довольствуясь региональными стандартами, связанными с вызывающим потоком. Для упрощения работы во многие типы добавлены перегруженные версии метода `ToString()`, вызывающие `IFormattable.ToString()` с аргументами по умолчанию:

```
int x = 1024;
var s1 = x.ToString();
var s2 = x.ToString("X");
var s3 = x.ToString(CultureInfo.InvariantCulture);
```

Любой тип может определять дополнительные методы для получения строкового представления данных. Например, в `DateTime` определены методы `ToLongDateString()`, `ToShortDateString()`, `ToLongTimeString()`, `ToShortDateString()`.

Для получения данных типа по строке обычно используются статические методы, которые по соглашению об именовании называются `Parse()` и `TryParse()`. Если преобразование из строки невозможно, метод `Parse()` генерирует исключение, а `TryParse()` возвращает значение `false`. Методы `Parse()` и `TryParse()` есть во всех примитивных типах, классе `Enum`, в типах для работы со временем и во многих других типах. Обычно эти методы имеют перегруженные версии, принимающие в качестве аргумента поставщик формата. Некоторые типы дополнительно перегружают `Parse()` и `TryParse()` для более тонкой настройки преобразования:

```
int x = Int32.Parse("1024");
int z = Int32.Parse("A03", NumberStyles.HexNumber);
int y;
if (Int32.TryParse("4201", out y))
{
    Console.WriteLine("Success");
}
DateTime dt = DateTime.Parse("13/01/2000 16:45:06");
```

Платформа .NET способна поддерживать различные текстовые кодировки и наборы символов. Для этого используется базовый класс `System.Text.Encoding`

и наследники этого класса – конкретные кодировки. Чтобы получить объект-кодировку можно использовать статический метод `Encoding.GetEncoding()` или статические свойства для популярных кодировок:

```
// используется имя кодировки по стандарту IANA (см. сайт iana.org)
Encoding utf8 = Encoding.GetEncoding("utf-8");

// распространённую кодировку можно получить через свойство
Encoding ascii = Encoding.ASCII;
```

Основными методами каждого объекта-кодировки являются `GetBytes()` и `GetString()`. Первый метод нужен для перевода строки или массива символов в массив байтов (коды символов), второй метод делает обратное преобразование:

```
Encoding ascii = Encoding.ASCII;
byte[] asciiBytes = ascii.GetBytes("Sample text");
string s = ascii.GetString(asciiBytes);
```

5. Сравнение для выяснения равенства

Платформа .NET и язык C# предлагают несколько стандартных протоколов для выяснения равенства объектов. Наиболее общий подход при реализации проверки равенства заключается в переопределении виртуального метода `Equals()` класса `Object`. Базовая версия этого метода использует равенство ссылок. Тип `ValueType` перекрывает `Equals()`, чтобы реализовать равенство по значению, то есть проверку на совпадение всех соответствующих полей двух переменных типа значения.

Основными причинами перекрытия `Equals()` в пользовательском типе являются особая семантика равенства, перенос на ссылочный тип равенства по значению, необходимость ускорения проверки на равенство для типа значения. Перекрытая версия `Equals()` должна удовлетворять следующим требованиям:

1. `x.Equals(x) == true`.
2. `x.Equals(y) == y.Equals(x)`.
3. `(x.Equals(y) && y.Equals(z)) == true` ⟹ `x.Equals(z) == true`.
4. Вызовы `x.Equals(y)` возвращают одинаковое значение до тех пор, пока `x` и `y` остаются неизменными.
5. `x.Equals(null) == false`, если `x != null`.
6. Метод `Equals()` не должен генерировать исключений.

Так как метод `Equals()` класса `Object` принимает в качестве аргумента объект, для типов значений при вызове метода происходит операция упаковки. Чтобы избежать этого, тип может дополнительно к перекрытию `Equals()` реализовать интерфейс `IEquatable<T>`:

```
public interface IEquatable<T>
{
    bool Equals(T other);
}
```

Рассмотрим пример неизменяемой структуры `Area`, в которой выполнено перекрытие метода `Equals()`, реализация интерфейса `IEquatable<T>` и перекрыта операция проверки равенства.

```
public struct Area : IEquatable<Area>
{
    public readonly int Measure1;
    public readonly int Measure2;

    public Area(int m1, int m2)
    {
        Measure1 = Math.Min(m1, m2);
        Measure2 = Math.Max(m1, m2);
    }

    public override bool Equals(object other)
    {
        if (other is Area)
        {
            return Equals((Area) other);
        }
        return false;

        // альтернативная реализация (подходит для типов значений)
        // Area? otherArea = other as Area?;
        // return otherArea.HasValue && Equals(otherArea.Value);
    }

    public bool Equals(Area other)    // реализация IEquatable<Area>
    {
        return Measure1 == other.Measure1
            && Measure2 == other.Measure2;
    }

    public override int GetHashCode()
    {
        return Measure1 * 31 + Measure2;
    }

    public static bool operator ==(Area a1, Area a2)
    {
        return a1.Equals(a2);
    }

    public static bool operator !=(Area a1, Area a2)
    {
        return !a1.Equals(a2);
    }
}
```

Базовые типы значений платформы .NET (включая структуры для представления времени) реализуют интерфейс `IEquatable<T>` и перекрывают операции `==` и `!=`. Тип `string` дополнительно содержит перегруженную версию `Equals()`, позволяющую выполнить сравнение, учитывающее региональные стандарты или нечувствительное к регистру. Эта версия принимает в качестве аргумента один из элементов перечисления `StringComparison`:

`CurrentCulture` – сравнение в алфавите текущего регионального стандарта;

`CurrentCultureIgnoreCase` – сравнение в алфавите текущего регионального стандарта без учёта регистра символов;

`InvariantCulture` – сравнение в алфавите инвариантной культуры;

`InvariantCultureIgnoreCase` – сравнение в алфавите инвариантной культуры без учёта регистра;

`Ordinal` – *порядковое сравнение*, при котором символы интерпретируются как числа (коды в UTF-16);

`OrdinalIgnoreCase` – порядковое сравнение без учёта регистра.

Если создатель типа не реализовал эффективный метод проверки равенства, этот недостаток можно восполнить, применив подходящий *подключаемый интерфейс*. Интерфейсы `System.Collections.Generic.IEqualityComparer<T>` и `System.Collections.IEqualityComparer` позволяют организовать проверку объектов на равенство и вычисление хэш-кода объекта:

```
public interface IEqualityComparer
{
    bool Equals(object x, object y);
    int GetHashCode(object obj);
}

public interface IEqualityComparer<in T>
{
    bool Equals(T x, T y);
    int GetHashCode(T obj);
}
```

Желательно, чтобы во вспомогательном типе для проверки равенства были реализованы обе версии интерфейса `IEqualityComparer`. Существует абстрактный класс `System.Collections.Generic.EqualityComparer<T>`, реализующий интерфейсы `IEqualityComparer<T>` и `IEqualityComparer` и содержащий виртуальные методы `Equals()` и `GetHashCode()`. Можно наследовать от этого класса и заместить его виртуальные методы.

```
public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

```
public class LastFirstComparer : EqualityComparer<Customer>
{
    public override bool Equals(Customer x, Customer y)
    {
        return x.LastName == y.LastName &&
               x.FirstName == y.FirstName;
    }

    public override int GetHashCode(Customer obj)
    {
        return (obj.LastName + ";" + obj.FirstName).GetHashCode();
    }
}
```

Статическое свойство `EqualityComparer<T>.Default` возвращает экземпляр `EqualityComparer<T>` для типа `T`. Этот экземпляр использует метод `Equals(T)` либо метод `Equals(object)` в зависимости от того, реализует ли тип `T` интерфейс `IEquatable<T>`.

Ранее указывалось, что по умолчанию структуры реализуют равенство по значению: две структуры равны, если равны значения всех их соответствующих элементов. Интерфейс `System.Collections.IStructuralEquatable` позволяет перенести данную семантику на другие типы. На платформе .NET этот интерфейс явно реализуют любые массивы и кортежи (классы вида `System.Tuple<>`).

```
public interface IStructuralEquatable
{
    bool Equals(object other, IEqualityComparer comparer);
    int GetHashCode(IEqualityComparer comparer);
}
```

Передаваемый в метод `Equals()` объект `comparer` применяется к каждому индивидуальному элементу в составном объекте. Следующий пример демонстрирует использование `IStructuralEquatable`.

```
string[] low = "the quick brown fox".Split();
string[] high = "THE QUICK BROWN FOX".Split();
IStructuralEquatable se = low;
// переменная f равна true
bool f = se.Equals(high, StringComparer.InvariantCultureIgnoreCase);
```

6. Сравнение для выяснения порядка

Стандартным способом сравнения для выяснения порядка является реализация типом интерфейсов `IComparable` и `IComparable<T>`:

```
public interface IComparable
{
    int CompareTo(object other);
}
```

```
public interface IComparable<in T>
{
    int CompareTo(T other);
}
```

Оба интерфейса предоставляют одинаковый функционал. Универсальный интерфейс `IComparable<T>` избавляет от необходимости упаковки при работе с типами значений. Метод `x.CompareTo(y)` должен возвращать любое положительное число, если `x` «больше» `y`, отрицательное число, если `x` «меньше» `y`, и ноль, если `x` и `y` равны.

Базовые типы реализуют обе версии интерфейса `IComparable`. В типе `string` метод `CompareTo()` выполняет чувствительное к регистру сравнение строк с учётом региональных стандартов. В некоторых базовых типах дополнительно определён статический метод сравнения `Compare()`. Например, `string` имеет перегруженные версии `Compare()`, которые принимают или элемент перечисления `StringComparison`, или булев флаг (игнорирование регистра) и объект `CultureInfo`:

```
string s1 = "Strasse", s2 = "Straße";
// порядковое сравнение, i = -108
int i = string.Compare(s1, s2, StringComparison.Ordinal);
// j = 0, так как в немецком языке такие слова равны
int j = string.Compare(s1, s2, false, new CultureInfo("de-DE"));
```

Кроме реализации `IComparable` и `IComparable<T>` тип может дополнительно перегружать операции `<` и `>`. Перегрузка операций сравнения обычно выполняется, если результат операции не зависит от контекста выполнения[1].

```
bool isAfterDoomsday = DateTime.Now > new DateTime(2012, 12, 21);
```

Как и в случае проверки на равенство, сравнение для выяснения порядка можно выполнить при помощи подключаемых интерфейсов. Для порядкового сравнения предназначены интерфейсы `IComparer<T>` и `IComparer` (пространства имён `System.Collections.Generic` и `System.Collections`).

```
public interface IComparer
{
    int Compare(object x, object y);
}

public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

[1] Тип `string` не перегружает операции сравнения, так как результат может измениться в зависимости от текущих региональных стандартов.

Существует абстрактный класс System.Collections.Generic.Comparer<T>, облегчающий реализацию интерфейсов IComparer<T> и IComparer:

```
public class Wish
{
    public string Name { get; set; }
    public int Priority { get; set; }
}

public class PriorityComparer : Comparer<Wish>
{
    public override int Compare(Wish x, Wish y)
    {
        if (Equals(x, y))
        {
            return 0;
        }
        return x.Priority.CompareTo(y.Priority);
    }
}
```

Статическое свойство Comparer<T>.Default возвращает экземпляр Comparer<T> для типа T. Этот экземпляр использует метод CompareTo(T) либо метод CompareTo(object) в зависимости от того, реализует ли тип T интерфейс IComparable<T>.

Абстрактный класс System.StringComparer предназначен для сравнения строк с учётом языка и регистра. Этот класс реализует интерфейсы IComparer, IEqualityComparer, IComparer<string> и IEqualityComparer<string>. Некоторые неабстрактные наследники StringComparer доступны через его статические свойства. Например, свойство StringComparer.CurrentCulture возвращает объект, реализующий сравнение строк с учётом текущих региональных стандартов.

Интерфейс System.Collections.IStructuralComparable позволяет провести порядковое сравнение составных объектов, выполняя сравнение отдельных элементов этих объектов. На платформе .NET этот интерфейс явно реализуют массивы и кортежи.

```
public interface IStructuralComparable
{
    int CompareTo(object other, IComparer comparer);
}
```

7. Жизненный цикл объектов

Все типы платформы .NET делятся на ссылочные типы и типы значений. Переменные типов значений создаются в стеке, их время жизни ограничено тем блоком кода, в котором они объявляются. Например, если переменная типа значения объявлена в некотором методе, то после выхода из метода память в стеке, занимаемая переменной, автоматически освободится. Переменные ссылочного

типа (объекты) располагаются в динамической памяти – *управляемой куче* (managed heap). Размещение объектов в управляемой куче происходит последовательно. Для этого CLR поддерживает указатель на свободное место в куче, перемещая его на соответствующее количество байтов после выделения памяти очередному объекту.

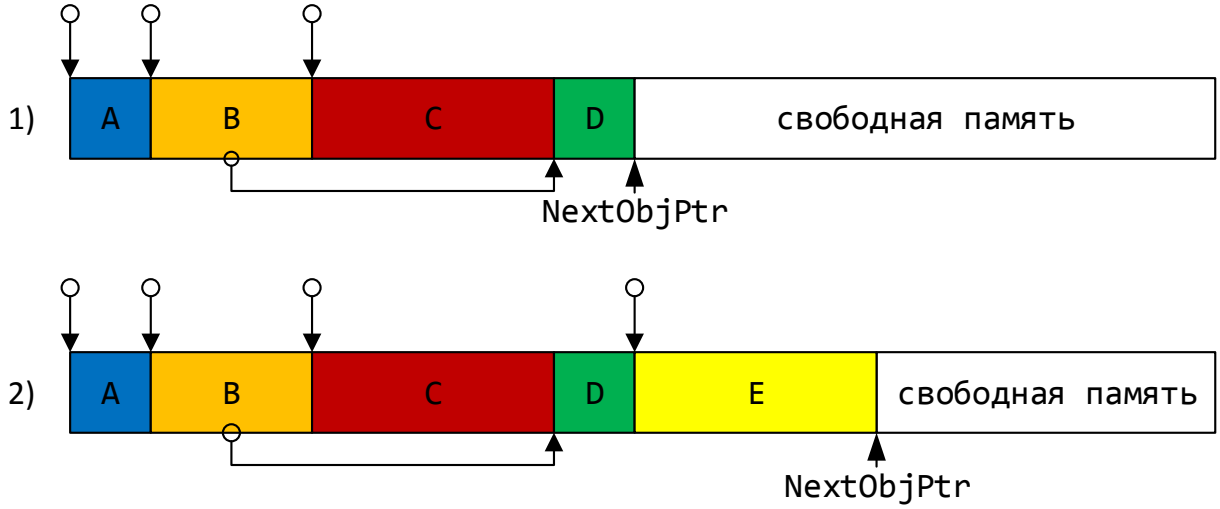

Рис. 1. Управляемая куча до (1) и после (2) выделения памяти под объект `E`.

7.1. Алгоритм сборки мусора

Если новый объект требует для размещения больше памяти, чем имеющийся свободный объем, CLR запускает процесс, называемый *сборка мусора*[1] (garbage collection). На первом этапе сборки мусора строится *граф используемых объектов*. Отправными точками в построении графа являются *корневые объекты*. Это объекты следующих категорий:

– локальная переменная или аргумент выполняемого метода (а также всех методов в стеке вызова);

– статическое поле;

– объект в *очереди завершения* (этот термин будет разъяснён позже).

При помощи графа используемых объектов выясняется реально занимаемая этими объектами память. Затем происходит *дефрагментация кучи* – используемые объекты перераспределяются так, чтобы занимаемая ими память составляла единый блок в начале кучи. После этого сборка мусора завершается, и новый объект размещается в управляемой куче.

[1] Компилятор C# генерирует код, принудительно запускающий сборку мусора при окончании работы программы.

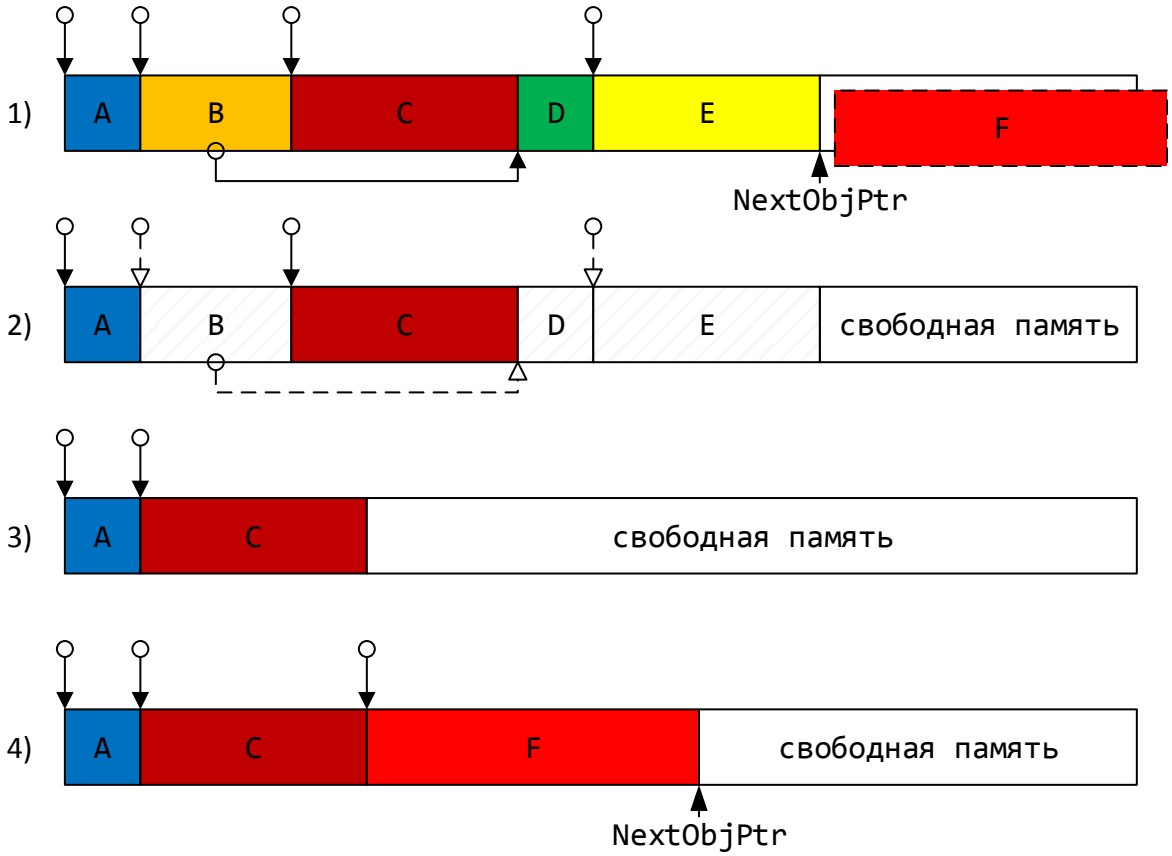

Рис. 2. Фазы сборки мусора: 1) инициирование, 2) выявление используемых объектов, 3) дефрагментация, 4) размещение объекта.

При размещении и удалении объектов CLR использует ряд оптимизаций. Во-первых, объекты размером 85000 и более байтов размещаются в отдельной управляемой *куче больших объектов* (Large Object Heap). При сборке мусора данная куча не дефрагментируется, так как копирование больших блоков памяти снижает производительность. Во-вторых, в управляемой куче для малых объектов выделяется три поколения – Gen0, Gen1 и Gen2. Вначале все объекты относятся к поколению Gen0. После первой сборки мусора «пережившие» её объекты переходят в поколение Gen1. В дальнейшем сборка мусора будет работать с объектами поколения Gen1, только если освобождение памяти в Gen0 дало неудовлетворительный результат. Если в Gen1 произошла сборка мусора, то «пережившие» её объекты переходят в поколение Gen2. Отметим, что куча больших объектов всегда рассматривается как куча объектов поколения Gen2.

Сборщик мусора представлен статическим классом `System.GC`, который обладает несколькими полезными методами (приведён неполный список):

1. `Collect()` – вызывает принудительную сборку мусора в программе.
2. `GetGeneration()` – возвращает номер поколения для указанного объекта;
3. `SuppressFinalize()` – подавляет вызов финализатора для объекта;
4. `WaitForPendingFinalizers()` – приостанавливает текущий поток выполнения до тех пор, пока не будут выполнены все финализаторы освобождаемых объектов.

7.2. Финализаторы и интерфейс IDisposable

Обсудим автоматическую сборку мусора с точки зрения программиста, разрабатывающего некий класс. С одной стороны, такой подход имеет свои преимущества. В частности, практически исключаются случайные утечки памяти, которые могут вызвать «забытые» объекты. Однако объект может в процессе создания или работы резервировать неуправляемые системные ресурсы, которые нужно освободить после того, как объект перестаёт использоваться. Примерами таких ресурсов являются открытый операционной системой файл или сетевое подключение. Платформа .NET предлагает подход, обеспечивающий автоматическое освобождение ресурсов при уничтожении объекта. Тип `System.Object` содержит виртуальный метод `Finalize()`. Класс (но не структура!) может переопределить этот метод для освобождения неуправляемых ресурсов. Объект такого класса обрабатывается особо при размещении в куче и при сборке мусора. При размещении ссылка на объект запоминается в специальной внутренней структуре CLR. При сборке мусора эта ссылка перемещается в *очередь завершения* (freachable queue). Затем в отдельном программном потоке у объектов из очереди завершения происходит вызов метода `Finalize()`, после этого ссылка на объект удаляется из очереди завершения.

Язык C# не позволяет явно переопределить в пользовательском классе метод `Finalize()`. Вместо этого в классе описывается специальный *финализатор*. Имя финализатора имеет вид *~имя-класса*`()`, он не имеет параметров и модификаторов доступа (считается, что у него модификатор доступа `protected`). При наследовании в финализатор класса-наследника автоматически подставляется вызов финализатора класса-предка.

Приведём пример класса с финализатором:

```
public class ClassWithFinalizer
{
    public void DoSomething()
    {
        Console.WriteLine("I am working...");
    }

    ~ClassWithFinalizer()
    {
        // здесь должен быть код освобождения неуправляемых ресурсов
        Console.WriteLine("Bye!");
    }
}
```

Вызов метода `Finalize()` является недетерминированным. Для освобождения управляемых ресурсов (т. е. ресурсов платформы .NET) программист может описать в классе некий метод, который следует вызывать вручную, когда ресурс больше не нужен. Для унификации данного решения предлагается интерфейс

`IDisposable` с единственным методом `Dispose()`. Если класс или структура реализуют этот интерфейс, `Dispose()` содержит код освобождения управляемых ресурсов.

```
public class ClassWithDispose : IDisposable
{
    public void DoSomething()
    {
        Console.WriteLine("I am working...");
    }

    public void Dispose()
    {
        // здесь должен быть код освобождения управляемых ресурсов
        Console.WriteLine("Bye!");
    }
}
```

Язык C# имеет специальный оператор `using`, который гарантирует вызов метода `Dispose()` для заданных переменных в конце блока кода. Синтаксис оператора `using` следующий:

```
using (получение-ресурса) вложенный-оператор
```

Здесь *получение-ресурса* означает один из вариантов.

1. Объявление и инициализацию локальной переменой (или списка переменных). Тип переменной должен реализовывать `IDisposable`. Такая переменная в блоке `using` доступна только для чтения.
2. Выражение, значение которого имеет тип, реализующий `IDisposable`.

Приведём пример использования оператора `using`:

```
using (ClassWithDispose x = new ClassWithDispose())
{
    x.DoSomething();
    // компилятор C# поместит сюда вызов x.Dispose()
}
```

7.3. Слабые ссылки

Слабая ссылка (weak reference) – особый вид ссылки на объект в системах со сборкой мусора. Если на объект имеются только слабые ссылки, он рассматривается алгоритмом сборки мусора как подлежащий удалению.

В .NET слабые ссылки представлены классами `System.WeakReference` и `System.WeakReference<T>`. Конструктор класса принимает объект, на который создаётся слабая ссылка. Свойство `Target` указывает на этот объект или имеет значение `null`, если объект был удалён сборщиком мусора.

```
var weak = new WeakReference(new StringBuilder("Test"));
```

```
if (weak.IsAlive)
{
    Console.WriteLine(weak.Target);        // Test
}
GC.Collect();
Console.WriteLine(weak.Target == null);    // True
```

Одно из применений слабых ссылок – построение кэшей больших объектов. При кэшировании формируется слабая ссылка на объект. При доступе проверяется свойство `Target`. Если оно равно `null`, объект повторно кэшируется.

8. Перечислители и итераторы

Рассмотрим стандартный код, работающий с массивом. Вначале массив инициализируется, затем печатаются все его элементы:

```
int[] data = {1, 2, 4, 8};
foreach (int item in data)
{
    Console.WriteLine(item);
}
```

Данный код работоспособен по двум причинам. Во-первых, любой массив относится к перечисляемым типам. Во-вторых, оператор `foreach` знает, как действовать с объектами перечисляемых типов. *Перечисляемый тип* (enumerable type) – это тип с экземплярным методом `GetEnumerator()`, возвращающим перечислитель. *Перечислитель* (enumerator) – объект, обладающий свойством `Current`, представляющим текущий элемент набора, и методом `MoveNext()` для перемещения к следующему элементу. Оператор `foreach` получает перечислитель, вызывая метод `GetEnumerator()`, а затем использует `MoveNext()` и `Current` для итерации по набору.

```
// семантика работы оператора foreach из предыдущего примера
int[] data = {1, 2, 4, 8};

var enumerator = data.GetEnumerator();
while (enumerator.MoveNext())
{
    int item = (int) enumerator.Current;
    Console.WriteLine(item);
}
```

В дальнейших примерах параграфа будет использоваться класс `Shop`, представляющий «магазин», который хранит некие «товары».

```
public class Shop
{
    private string[] _items = new string[0];
```

```
        public int ItemsCount
        {
            get { return _items.Length; }
        }

        public void AddItem(string item)
        {
            Array.Resize(ref _items, ItemsCount + 1);
            _items[ItemsCount - 1] = item;
        }

        public string GetItem(int index)
        {
            return _items[index];
        }
    }
```

Пусть требуется сделать класс `Shop` перечисляемым. Для этого существует три способа:

1. Реализовать интерфейсы `IEnumerable` и `IEnumerator`.
2. Реализовать интерфейсы `IEnumerable<T>` и `IEnumerator<T>`.
3. Способ, при котором стандартные интерфейсы не применяются.

Интерфейсы `IEnumerable` и `IEnumerator` описаны в пространстве имён `System.Collections`:

```
    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }

    public interface IEnumerator
    {
        object Current { get; }
        bool MoveNext();
        void Reset();
    }
```

Свойство для чтения `Current` представляет текущий объект набора. Для обеспечения универсальности это свойство имеет тип `object`. Метод `MoveNext()` выполняет перемещение на следующую позицию в наборе. Этот метод возвращает значение `true`, если дальнейшее перемещение возможно. Предполагается, что `MoveNext()` нужно вызвать и для получения первого элемента, то есть начальная позиция – «перед первым элементом». Метод `Reset()` сбрасывает позицию в начальное состояние.

Добавим поддержку интерфейсов `IEnumerable` и `IEnumerator` в класс `Shop`. Обратите внимание, что для этого используется вложенный класс, реализующий интерфейс `IEnumerator`.

```
public class Shop : IEnumerable
{
    // опущены элементы ItemsCount, AddItem(), GetItem()

    private class ShopEnumerator : IEnumerator
    {
        private readonly string[] _data; // локальная копия данных
        private int _position = -1;      // текущая позиция в наборе

        public ShopEnumerator(string[] values)
        {
            _data = new string[values.Length];
            Array.Copy(values, _data, values.Length);
        }

        public object Current
        {
            get { return _data[_position]; }
        }

        public bool MoveNext()
        {
            if (_position < _data.Length - 1)
            {
                _position++;
                return true;
            }
            return false;
        }

        public void Reset()
        {
            _position = -1;
        }
    }

    public IEnumerator GetEnumerator()
    {
        return new ShopEnumerator(_items);
    }
}
```

Теперь класс `Shop` можно использовать следующим образом:

```
var shop = new Shop();
shop.AddItem("computer");
shop.AddItem("monitor");

foreach (string s in shop)
    Console.WriteLine(s);
```

При записи цикла `foreach` объявляется переменная, тип которой совпадает с типом элемента коллекции. Так как свойство `IEnumerator.Current` имеет тип `object`, то на каждой итерации выполняется приведение этого свойства к типу переменной цикла[1]. Это может повлечь ошибки времени выполнения. Избежать ошибок помогает реализация перечисляемого типа при помощи универсальных интерфейсов `IEnumerable<T>` и `IEnumerator<T>`:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

Универсальные интерфейсы `IEnumerable<T>` и `IEnumerator<T>` наследуются от обычных версий. У интерфейса `IEnumerator<T>` типизированное свойство `Current`. Тип, реализующий интерфейс `IEnumerable<T>`, должен содержать две версии метода `GetEnumerator()`. Обычно для `IEnumerable.GetEnumerator()` применяется явная реализация.

```
public class Shop : IEnumerable<string>
{
    // опущены элементы ItemsCount, AddItem(), GetItem()

    private class ShopEnumerator : IEnumerator<string>
    {
        private readonly string[] _data;
        private int _position = -1;

        public ShopEnumerator(string[] values)
        {
            _data = new string[values.Length];
            Array.Copy(values, _data, values.Length);
        }

        public string Current
        {
            get { return _data[_position]; }
        }

        object IEnumerator.Current
        {
            get { return _data[_position]; }
        }
```

[1] Если бы использовалось `foreach (var s in shop)`, то типом `s` был бы `object`.

```
        public bool MoveNext()
        {
            if (_position < _data.Length - 1)
            {
                _position++;
                return true;
            }
            return false;
        }

        public void Reset()
        {
            _position = -1;
        }

        public void Dispose() { /* пустая реализация */ }
    }

    public IEnumerator<string> GetEnumerator()
    {
        return new ShopEnumerator(_items);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Возможна (хотя и нетипична) реализация перечисляемого типа без использования стандартных интерфейсов:

```
public class Shop
{
    // опущены элементы ItemsCount, AddItem(), GetItem()

    public ShopEnumerator GetEnumerator()
    {
        return new ShopEnumerator(_items);
    }
}

public class ShopEnumerator
{
    // реализация соответствует первому примеру,
    // где ShopEnumerator – вложенный класс
    public string Current { get { . . . } }
    public bool MoveNext() { . . . }
}
```

Во всех предыдущих примерах для перечислителя создавался пользовательский класс. Существуют альтернативные подходы к реализации перечислителя. Так как перечисляемый тип обычно хранит свои данные в стандартной коллекции или массиве, то обычно достаточно вернуть перечислитель этой коллекции:

```
public class Shop : IEnumerable
{
    // опущены элементы ItemsCount, AddItem(), GetItem()

    public IEnumerator GetEnumerator()
    {
        return _items.GetEnumerator();      // перечислитель массива
    }
}
```

Создать перечислитель можно при помощи итератора. *Итератор* (iterator) – это операторный блок, который порождает упорядоченную последовательность значений. Итератор отличает присутствие одного или нескольких операторов `yield`. Оператор `yield return` *выражение* возвращает следующее значение последовательности, а оператор `yield break` прекращает генерацию последовательности. Итераторы могут использоваться в качестве тела метода, если тип метода – один из интерфейсов `IEnumerator`, `IEnumerator<T>`, `IEnumerable`, `IEnumerable<T>`.

Реализуем метод `Shop.GetEnumerator()` при помощи итератора.

```
public class Shop : IEnumerable<string>
{
    // опущены элементы ItemsCount, AddItem(), GetItem()

    public IEnumerator<string> GetEnumerator()
    {
        foreach (var s in _items)
        {
            yield return s;
        }
    }
}
```

Как видим, код заметно упростился. Элементы коллекции перебираются в цикле, и для каждого вызывается `yield return s`. Но достоинства итераторов этим не ограничиваются. В следующем примере в класс `Shop` добавляется метод, позволяющий перебрать коллекцию в обратном порядке.

```
public class Shop : IEnumerable<string>
{
    // опущены элементы, описанные ранее

    public IEnumerable<string> GetItemsReversed()
    {
```

```
            for (var i = _items.Length; i > 0; i--)
            {
                yield return _items[i - 1];
            }
        }
    }

    // пример использования
    foreach (var s in shop.GetItemsReversed())
    {
        Console.WriteLine(s);
    }
```

Итераторы реализуют концепцию *отложенных вычислений*. Каждое выполнение оператора `yield return` ведёт к выходу из метода и возврату значения. Но состояние метода, его внутренние переменные и позиция `yield return` запоминаются, чтобы быть восстановленными при следующем вызове.

Поясним концепцию отложенных вычислений на примере. Пусть имеется класс `Helper` с итератором `GetNumbers()`.

```
public static class Helper
{
    public static IEnumerable<int> GetNumbers()
    {
        int i = 0;
        while (true) yield return i++;
    }
}
```

Кажется, что вызов метода `GetNumbers()` приведёт к «зацикливанию» программы. Однако использование итераторов обеспечивает этому методу следующее поведение. При первом вызове `GetNumbers()` вернёт значение `i = 0`, и состояние метода (значение переменной `i`) будет зафиксировано. При следующем вызове метод вернёт значение `i = 1` и снова зафиксирует своё состояние, и так далее. Таким образом, следующий код успешно печатает три числа:

```
foreach (var number in Helper.GetNumbers())
{
    Console.WriteLine(number);
    if (number == 2) break;
}
```

Рассмотрим ещё один пример итераторов. Пусть описан класс `Range`:

```
public class Range
{
    public int Low { get; set; }
    public int High { get; set; }
```

```
    public IEnumerable<int> GetNumbers()
    {
        for (int counter = Low; counter <= High; counter++)
        {
            yield return counter;
        }
    }
}
```

Используем класс `Range` следующим образом:

```
var range = new Range {Low = 0, High = 10};
var enumerator = range.GetNumbers();
foreach (int number in enumerator)
{
    Console.WriteLine(number);
}
```

На консоль будут выведены числа от 0 до 10. Интересно, что если изменить объект `range` *после* получения перечислителя `enumerator`, это повлияет на выполнение цикла `foreach`. Следующий код выводит числа от 0 до 5.

```
var range = new Range {Low = 0, High = 10};
var enumerator = range.GetNumbers();
range.High = 5;    // изменяем свойство объекта range
foreach (int number in enumerator)
{
    Console.WriteLine(number);
}
```

Возможности итераторов широко используются в технологии LINQ to Objects, которая будет описана далее.

9. Стандартные интерфейсы коллекций

Платформа .NET включает большой набор типов для предоставления стандартных коллекций – списков, множеств, словарей. Эти типы можно разделить на несколько категорий: базовые интерфейсы и вспомогательные классы, классы для коллекций-списков и словарей, набор классов для построения собственных коллекций. Типы сгруппированы в следующие пространства имён:

1. `System.Collections` – коллекции, в которых элемент коллекции представлен как `object` (*слаботипизированные коллекции*).
2. `System.Collections.Specialized` – специальные коллекции.
3. `System.Collections.Generic` – универсальные классы и интерфейсы коллекций.
4. `System.Collections.ObjectModel` – базовые и вспомогательные типы, которые могут применяться для построения пользовательских коллекций.

5. System.Collections.Concurrent – коллекции для использования в многопоточных приложениях.

Опишем набор интерфейсов, реализуемых практически всеми типами коллекций. Основу набора составляют интерфейсы IEnumerable<T> и IEnumerable. Они отражают фундаментальное свойство любой коллекции – возможность перечислить её элементы. Слаботипизированные словари реализуют интерфейс IDictionaryEnumerator для перебора пар «ключ-значение» (DictionaryEntry – вспомогательная структура, у которой определены свойства Key и Value).

```
public interface IDictionaryEnumerator : IEnumerator
{
    DictionaryEntry Entry { get; }
    object Key { get; }
    object Value { get; }
}
```

Интерфейс ICollection предназначен для коллекций, запоминающих число хранимых элементов. Этот интерфейс определяет свойство Count, а также метод для копирования коллекции в массив и свойства для синхронизации коллекции при многопоточном использовании.

```
public interface ICollection : IEnumerable
{
    // метод
    void CopyTo(Array array, int index);
    // свойства
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
}
```

Универсальный интерфейс ICollection<T> также поддерживает свойство для количества элементов. Кроме этого, он предоставляет методы для добавления и удаления элементов, копирования элементов в массив, поиска элемента.

```
public interface ICollection<T> : IEnumerable<T>
{
    // методы
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool Remove(T item);

    // свойства
    int Count { get; }
    bool IsReadOnly { get; }
}
```

Интерфейс `IList` описывает набор данных, которые проецируются на массив. Дополнительно к функциональности, унаследованной от `IEnumerable` и `ICollection`, интерфейс `IList` позволяет обращаться к элементу по индексу, добавлять, удалять и искать элементы.

```
public interface IList : ICollection
{
    // методы
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);

    // свойства
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
}
```

Возможности интерфейса `IList<T>` тождественны возможностям `IList`.

```
public interface IList<T> : ICollection<T>
{
    // методы
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);

    // свойство
    T this[int index] { get; set; }
}
```

В качестве примера работы с интерфейсом `IList<T>` рассмотрим метод `Permutate()`, генерирующий все перестановки указанного набора данных.

```
public static IEnumerable<IList<T>> Permutate<T>(IList<T> sequence,
                                                 int count)
{
    if (count == 1)
    {
        yield return sequence;
    }
    else
    {
        for (var i = 0; i < count; i++)
        {
            foreach (var perm in Permutate(sequence, count - 1))
```

```
                {
                    yield return perm;
                }

                // циклический сдвиг sequence
                T last = sequence[count - 1];
                sequence.RemoveAt(count - 1);
                sequence.Insert(0, last);
            }
        }
    }
```

Интерфейсы `IDictionary` и `IDictionary<TKey, TValue>` определяют протокол взаимодействия для коллекций-словарей (`KeyValuePair<TKey, TValue>` – это вспомогательная структура, у которой определены свойства `Key` и `Value`).

```
public interface IDictionary : ICollection
{
    // методы
    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key);

    // свойства
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[object key] { get; set; }
    ICollection Keys { get; }        // все ключи словаря
    ICollection Values { get; }      // все значения словаря
}

public interface IDictionary<TKey, TValue> :
                            ICollection<KeyValuePair<TKey, TValue>>
{
    // методы
    void Add(TKey key, TValue value);
    bool ContainsKey(TKey key);
    bool Remove(TKey key);
    bool TryGetValue(TKey key, out TValue value);

    // свойства
    TValue this[TKey key] { get; set; }
    ICollection<TKey> Keys { get; }
    ICollection<TValue> Values { get; }
}
```

Для работы с коллекциями-множествами предназначен интерфейс `ISet<T>`. Его набор методов отражает типичные операции для множеств.

```
public interface ISet<T> : ICollection<T>
{
    bool Add(T item);
    void ExceptWith(IEnumerable<T> other);
    void IntersectWith(IEnumerable<T> other);
    bool IsProperSubsetOf(IEnumerable<T> other);
    bool IsProperSupersetOf(IEnumerable<T> other);
    bool IsSubsetOf(IEnumerable<T> other);
    bool IsSupersetOf(IEnumerable<T> other);
    bool Overlaps(IEnumerable<T> other);
    bool SetEquals(IEnumerable<T> other);
    void SymmetricExceptWith(IEnumerable<T> other);
    void UnionWith(IEnumerable<T> other);
}
```

Для взаимодействия с коллекциями, предназначенными только для чтения, в платформе .NET версии 4.5 введён интерфейс `IReadOnlyList<T>`. Он позволяет узнать число хранимых элементов и получить элемент по индексу.

```
public interface IReadOnlyList<out T> : IEnumerable<T>
{
    // свойства
    int Count { get; }
    T this[int index] { get; }
}
```

Рис. 3 демонстрирует связи между интерфейсами коллекций.

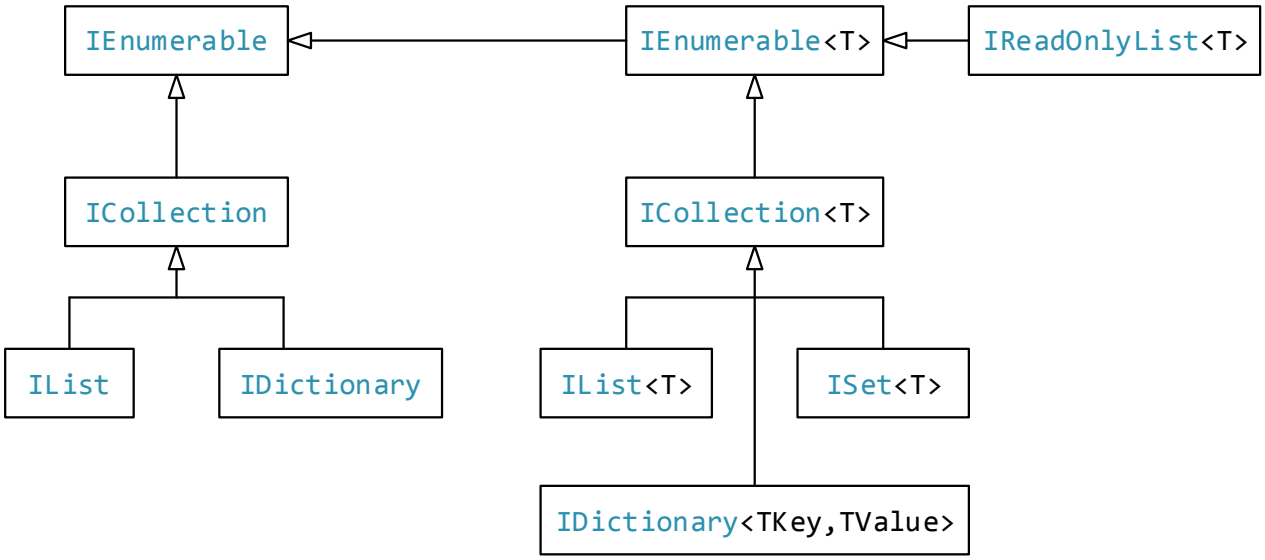

Рис. 3. Стандартные интерфейсы коллекций.

10. Массивы и класс System.Array

Класс `System.Array` является базовым классом для любого массива. Любой массив реализует интерфейсы `IStructuralEquatable`, `IStructuralComparable`, `IList` и `IList<T>`, причём первые три интерфейса реализованы явно (методы

`Add()` и `Remove()` генерируют исключение в случае коллекции фиксированной длины, которой является массив). В табл. 6 описаны экземплярные элементы любого массива, а табл. 7 содержит статические методы класса `Array`.

Таблица 6

Экземплярные элементы массива

Имя элемента	Описание
`Rank`	Свойство для чтения, возвращает размерность массива
`Length`	Свойство для чтения, возвращает общее число элементов массива
`LongLength`	Свойство для чтения, число элементов. Имеет тип `long`
`CopyTo()`	Метод копирует фрагмент массива в другой массив
`GetLength()`	Метод возвращает число элементов в указанном измерении
`GetLowerBound()`	Метод возвращает нижнюю границу для указанного измерения
`GetUpperBound()`	Метод возвращает верхнюю границу для указанного измерения
`GetValue()`	Метод возвращает значение элемента с указанными индексами
`SetValue()`	Метод устанавливает значение элемента с указанными индексами

Таблица 7

Статические методы класса `System.Array`

Имя метода	Описание
`Sort()`	Сортирует массив, переданный в качестве аргумента (возможно, с применением собственного объекта для сравнения элементов)
`BinarySearch()`	Поиска элемента в отсортированном массиве
`IndexOf()`, `LastIndexOf()`	Возвращается индекс первого (последнего) вхождения элемента в одномерный массив или фрагмент массива. Если элемент в массив не входит, возвращается значение, на единицу меньше нижней границы индекса
`Exists()`	Определяет, содержит ли массив хотя бы один элемент, удовлетворяющий предикату, который задан аргументом метода
`Find()`	Возвращает первый элемент, удовлетворяющий предикату, который задан аргументом метода
`FindLast()`	Возвращает первый элемент с конца массива, удовлетворяющий предикату, который задан аргументом метода
`FindAll()`	Возвращает все элементы, удовлетворяющие предикату, который задан аргументом метода
`FindIndex()`	Возвращает индекс первого вхождения элемента, удовлетворяющего предикату, заданному как аргумент метода
`FindLastIndex()`	Возвращает индекс последнего вхождения элемента, удовлетворяющего предикату, заданному как аргумент метода
`ConvertAll()`	Конвертирует массив одного типа в массив другого типа
`ForEach()`	Выполняет указанное действие для всех элементов массива
`TrueForAll()`	Возвращает `true`, если заданный предикат верен для всех элементов
`Clear()`	Устанавливает для массива или его части значение по умолчанию для типа элементов
`Reverse()`	Меняет порядок элементов в одномерном массиве или его части на противоположный
`AsReadOnly()`	Создаёт на основе массива коллекцию, не допускающую модификации своих элементов (read-only collection)
`CreateInstance()`	Создаёт экземпляр массива любого типа, размерности и длины

Окончание табл. 7

Имя метода	Описание
`Copy()`	Копирует раздел одного массива в другой массив, выполняя приведение типов
`ConstrainedCopy()`	Метод подобен `Copy()`, но если приведение типов для элементов неудачно, выполняется отмена операции копирования
`Resize()`	Позволяет изменить размер массива

Рассмотрим несколько примеров использования методов массива. В первом примере создадим массив и выполним работу с его элементами, не применяя традиционный синтаксис C#.

```
Array a = Array.CreateInstance(typeof (string), 2);    // тип, длина
a.SetValue("hi", 0);                  // a[0] = "hi";
a.SetValue("there", 1);               // a[1] = "there";
string s = (string) a.GetValue(0);    // s = a[0];
```

Метод `CreateInstance()` может создать массив любой размерности с указанным диапазоном изменения индексов:

```
// b – это массив int[-5..4, 100..119]
var b = Array.CreateInstance(typeof (int),     // тип элемента
                             new[] {10, 20},   // длины размерностей
                             new[] {-5, 100}); // нижние границы
b.SetValue(25, -3, 110);
int x = (int) b.GetValue(-2, 115);
```

Группа статических методов класса `Array` позволяет выполнить сортировку и поиск данных в массиве. Методы поиска могут использовать заданные предикаты, а сортировка – выполняться по заданному критерию сравнения.

```
int[] a = {10, 3, 5, -7, 0, 20, 10, 4};

// поиск элемента по предикату
int b = Array.Find(a, x => x > 6);

// поиск всех элементов
int[] c = Array.FindAll(a, x => x > 6);

// действие над элементами
Array.ForEach(c, Console.WriteLine);

// проверка условия
bool flag = Array.TrueForAll(a, x => x > 0);

// сортировка
Array.Sort(a, (x, y) => x == y ? 0 : x > y ? -1 : 1);

// двоичный поиск
int pos = Array.BinarySearch(a, 3);                // двоичный поиск
```

Массивы допускают копирование элементов и изменение размера:

```
int[] a = {10, 3, 5, -7, 0, 20, 10, 4};
int[] b = new int[a.Length];
long[] c = new long[a.Length];
a.CopyTo(b, 0);
Array.Copy(a, c, a.Length);
Array.Resize(ref a, 40);
```

Заметим, что для быстрого копирования массивов с элементами типа значений можно использовать класс `System.Buffer`. Он оперирует байтами данных.

```
int[] a = {10, 3, 5, -7, 0, 20, 10, 4};
int[] b = new int[a.Length];
Buffer.BlockCopy(a, 3, b, 5, 10);    // смещение задано в байтах
```

11. Типы для работы с коллекциями-списками

Рассмотрим типы из базовой библиотеки платформы .NET, применяемые при работе с коллекциями со списковой семантикой.

Класс `List<T>` из пространства имён `System.Collections.Generic` – основной класс для представления наборов, которые допускают динамическое добавление элементов[1]. Для хранения данных набора используется внутренний массив. Класс `List<T>` реализует интерфейсы `IList<T>`, `IList`, `IReadOnlyList<T>`. В табл. 8 представлено описание `public`-элементов класса `List<T>`.

Таблица 8

Элементы класса `List<T>`

Элемент	Описание
Добавление и удаление элементов	
`Add()`	Добавление одного элемента
`AddRange()`	Добавление набора элементов
`Insert()`	Вставка элемента в заданную позицию
`InsertRange()`	Вставка набора элементов
`Remove()`	Удаление элемента
`RemoveAt()`	Удаление элемента на указанной позиции со сдвигом остальных
`RemoveRange()`	Удаление диапазона элементов
`RemoveAll()`	Удаление всех элементов, удовлетворяющих заданному предикату
Индексирование элементов	
`this[int index]`	Основной индексатор
`GetRange()`	Получение подсписка
`GetEnumerator()`	Получение перечислителя
Поиск и сортировка	
`BinarySearch()`	Поиск элемента в упорядоченном наборе
`IndexOf()`	Индекс первого вхождения своего аргумента в набор
`LastIndexOf()`	Индекс последнего вхождения своего аргумента в набор

[1] В пространстве имён `System.Collections` имеется слаботипизированный аналог класса `List<T>` – класс `ArrayList`.

Окончание табл. 8

Элемент	Описание
`Contains()`	Проверка, содержится ли указанный элемент в наборе
`Exists()`	Проверка, содержит ли набор элемент, удовлетворяющий предикату
`Find()`	Возвращает первый элемент, удовлетворяющий предикату, который задан как аргумент метода
`FindLast()`	Возвращает первый элемент с конца набора, удовлетворяющий предикату, который задан как аргумент метода
`FindAll()`	Возвращает все элементы набора, удовлетворяющие предикату
`FindIndex()`	Возвращает индекс первого вхождения элемента, удовлетворяющего предикату, который задан как аргумент метода
`FindLastIndex()`	Возвращает индекс последнего вхождения элемента, удовлетворяющего предикату, который задан как аргумент метода
`TrueForAll()`	Возвращает `true`, если заданный предикат верен для всех элементов
`Sort()`	Сортировка набора (возможно, с применением собственного объекта для сравнения элементов)
Экспорт и конвертирование элементов	
`ToArray()`	Преобразование набора в массив
`CopyTo()`	Копирование набора или его части в массив
`AsReadOnly()`	Преобразование набора в коллекцию только для чтения
`ConvertAll()`	Конвертирование набора одно типа в набор другого типа
Другие методы и свойства	
`Count`	Количество элементов в наборе
`Capacity`	Ёмкость набора
`TrimExcess()`	Усечение размера внутреннего массива до необходимой минимальной величины
`Clear()`	Очистка списка (удаление всех элементов)
`Reverse()`	Изменение порядка элементов на противоположный
`ForEach()`	Выполняет указанное действие для всех элементов списка

Класс `List<T>` имеет три конструктора. Первый из них – обычный конструктор без параметров. Второй конструктор позволяет создать набор на основе коллекции – производится копирование элементов коллекции в список. Третий конструктор принимает в качестве аргумента начальную ёмкость набора. *Ёмкость набора* (capacity) – это количество элементов набора, которое он способен содержать без увеличения размера внутреннего массива.

Следующий код демонстрирует использование некоторых свойств и методов класса `List<T>`. Обратите внимание, что для добавления элементов в созданный набор применяется возможность инициализации классов-коллекций.

```
List<string> words = new List<string> {"melon", "avocado"};
words.AddRange(new[] {"banana", "plum"});
words.Insert(0, "lemon");
words.InsertRange(0, new[] {"peach", "apple"});
words.Remove("melon");
words.RemoveAt(3);
words.RemoveAll(s => s.StartsWith("a"));
List<string> subset = words.GetRange(1, 2);
string[] wordsArray = words.ToArray();
```

```
List<int> lengths = words.ConvertAll<int>(s => s.Length);
```

На примере `List<T>` рассмотрим особенность, присущую использованию коллекций в языке C#. Если коллекция хранит структуры, то C# не позволяет изменить части структуры при помощи индексатора коллекции (так как индексатор является функцией, возвращающей *копию* элемента коллекции):

```
public struct Student
{
    public string Name { get; set; }
}

var list = new List<Student> {new Student {Name = "Ivanov"}};
list[0].Name = "Petrov";    // ошибка компиляции!
```

Класс `LinkedList<T>` служит для представления двусвязного списка. Такой список позволяет осуществлять вставку и удаление элемента без сдвига остальных элементов. Однако доступ к элементу по индексу требует прохода по списку. `LinkedList<T>` реализует интерфейсы `ICollection` и `ICollection<T>`. Каждый элемент двусвязного списка представлен объектом `LinkedListNode<T>`.

```
public sealed class LinkedListNode<T>
{
    public LinkedList<T> List { get; }
    public LinkedListNode<T> Next { get; }
    public LinkedListNode<T> Previous { get; }
    public T Value { get; set; }
}
```

При добавлении элемента можно указать, чтобы он помещался в начало списка, или в конец списка, или относительно существующего в списке элемента. Для этого класс `LinkedList<T>` содержит специальные методы:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst(T value);

public void AddLast(LinkedListNode<T> node);
public LinkedListNode<T> AddLast(T value);

public void AddAfter(LinkedListNode<T> node,
                     LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);

public void AddBefore(LinkedListNode<T> node,
                      LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);
```

Аналогичные методы существуют и для удаления элементов списка:

```
public void RemoveFirst();
public void RemoveLast();
public bool Remove(T value);
public void Remove(LinkedListNode<T> node);
```

Класс `LinkedList<T>` содержит свойства для числа элементов, для указания на первый и последний элемент. Имеются методы для поиска элементов.

Ниже приведён пример использования `LinkedList<T>`.

```
var tune = new LinkedList<string>();
tune.AddFirst("do");                        // do
tune.AddLast("so");                         // do - so
tune.AddAfter(tune.First, "re");            // do - re- so
tune.AddAfter(tune.First.Next, "mi");       // do - re - mi- so
tune.AddBefore(tune.Last, "fa");            // do - re - mi - fa- so
tune.RemoveFirst();                         // re - mi - fa - so
tune.RemoveLast();                          // re - mi - fa
var miNode = tune.Find("mi");
tune.Remove(miNode);                        // re - fa
tune.AddFirst(miNode);                      // mi - re - fa
```

Классы `Queue<T>` и `Stack<T>` реализуют структуры данных «очередь» и «стек» на основе массива[1]. Конструкторы этих классов, как и конструкторы класса `List<T>`, позволяют создать объект на основе другой коллекции, а также указать значение для ёмкости (но ёмкость не доступна в виде отдельного свойства). Элементы классов вполне предсказуемы и описаны в табл. 9 и табл. 10.

Таблица 9

Элементы класса `Queue<T>`

Элемент	Описание
`Clear()`	Очистка очереди (удаление всех элементов)
`Contains()`	Проверка, содержится ли указанный элемент в очереди
`CopyTo()`	Копирование очереди в массив
`Count`	Количество элементов (свойство только для чтения)
`Dequeue()`	Извлечение элемента из очереди
`Enqueue()`	Помещение элемента в очередь
`GetEnumerator()`	Получение перечислителя
`Peek()`	Чтение очередного элемента без его удаления из очереди
`ToArray()`	Преобразование очереди в массив
`TrimExcess()`	Усечение размера внутреннего массива до необходимой минимальной величины

[1] В пространстве имён `System.Collections` имеются слаботипизированные аналоги классов `Queue<T>` и `Stack<T>` – классы `Queue` и `Stack`.

Таблица 10

Элементы класса `Stack<T>`

Элемент	Описание
`Clear()`	Очистка стека (удаление всех элементов)
`Contains()`	Проверка, содержится ли указанный элемент в стеке
`CopyTo()`	Копирование стека в массив
`Count`	Количество элементов (свойство только для чтения)
`GetEnumerator()`	Получение перечислителя
`Peek()`	Чтение очередного элемента без его удаления из стека
`Pop()`	Извлечение элемента из стека
`Push()`	Помещение элемента в стек
`ToArray()`	Преобразование стека в массив
`TrimExcess()`	Усечение размера внутреннего массива до необходимой минимальной величины

12. Типы для работы с коллекциями-множествами

В библиотеке базовых классов платформы .NET имеются два класса для представления множеств (то есть коллекций, которые не содержат повторяющихся элементов) – `HashSet<T>` и `SortedSet<T>`. Оба класса реализуют интерфейс `ISet<T>`.

Класс `HashSet<T>` описывает множество, в котором вхождение элемента проверяется на основе хэш-кода. Конструкторы класса `HashSet<T>` позволяют создать множество на основе коллекции, а также указать объект, реализующий интерфейс `IEqualityComparer<T>` для проверки равенства элементов множества. Кроме реализации интерфейса `ISet<T>`, класс `HashSet<T>` содержит экземплярный метод `RemoveWhere()` для удаления элементов, удовлетворяющих заданному предикату. Статический метод `CreateSetComparer()` возвращает экземпляр класса, реализующего `IEqualityComparer<HashSet<T>>`.

Следующий пример показывает использование `HashSet<T>`:

```
var setOne = new HashSet<char>("the quick brown fox");
var setTwo = new HashSet<char>("jumps over the lazy dog");
setOne.IntersectWith(setTwo);                // результат = the uro
Console.WriteLine(setOne.Contains('t'));     // True
Console.WriteLine(setOne.Contains('j'));     // False
setTwo.RemoveWhere(c => c < 'k');
```

Класс `SortedSet<T>` – это множество, поддерживающее набор элементов в отсортированном порядке. Конструкторы класса `SortedSet<T>` позволяют создать множество на основе коллекции, а также указать объект, реализующий интерфейс `IComparer<T>` для проверки порядка элементов множества. Набор методов `SortedSet<T>` схож с набором методов класса `HashSet<T>`. Экземплярный метод `Reverse()` возвращает набор элементов в противоположном порядке, а метод `GetViewBetween()` возвращает фрагмент («окно») исходного множества между двумя элементами.

```
var setOne = new SortedSet<char>("the quick brown fox");
Console.WriteLine("{0} {1}", setOne.Min, setOne.Max);
IEnumerable<char> reversed = setOne.Reverse();
var setTwo = setOne.GetViewBetween('a', 'p');
Console.WriteLine("{0} {1}", setTwo.Min, setTwo.Max);
```

13. Типы для работы с коллекциями-словарями

Под термином *словарь* будем понимать коллекцию, которая хранит пары «ключ-значение» с возможностью доступа к элементам по ключу. Базовая библиотека платформы .NET предлагает несколько коллекций-словарей, как классических, так и с различными дополнительными возможностями.

Универсальный класс `Dictionary<TKey, TValue>` – классический словарь с возможностью указать тип для ключа и тип для значения[1]. Данный класс является одним из наиболее часто используемых классов-коллекций (наряду с классом `List<T>`). Класс `Dictionary<TKey, TValue>` реализует обе версии интерфейса `IDictionary` (обычную и универсальную). Пример использования класса приведён ниже.

```
// конструируем словарь и помещаем в него один элемент
// обратите внимание на синтаксис инициализации словаря
var d = new Dictionary<string, int> {{"One", 1}};

// помещаем элементы, используя индексатор и метод Add()
d["Two"] = 22;
d.Add("Three", 3);

// обновляем существующий элемент
d["Two"] = 2;

Console.WriteLine(d["Two"]);
Console.WriteLine(d.ContainsKey("One"));    // быстрая операция
Console.WriteLine(d.ContainsValue(3));      // медленная операция

int val;
if (!d.TryGetValue("ONE", out val))
{
    Console.WriteLine("No such value");
}

// различные способы перечисления словаря
foreach (var pair in d)
{
    Console.WriteLine(pair.Key + "; " + pair.Value);
}
```

[1] В пространстве имён `System.Collections` имеется слаботипизированный аналог класса `Dictionary<TKey,TValue>` – класс `Hashtable`.

```
foreach (string key in d.Keys)
{
    Console.WriteLine(key);
}
foreach (int value in d.Values)
{
    Console.WriteLine(value);
}
```

Словарь может работать с элементами любого типа, так как у любого объекта можно получить хэш-код и сравнить объекты на равенство, используя методы `GetHashCode()` и `Equals()`. Пользовательские типы могут переопределять данные методы, предоставляя их эффективную реализацию. Кроме этого, конструктору словаря можно передать объект, реализующий интерфейс `IEqualityComparer<TKey>`. Типичным примером использования такого подхода является конструирование словаря, обеспечивающего сравнение строк-ключей независимо от их регистра:

```
var comparer = StringComparer.OrdinalIgnoreCase;
var d = new Dictionary<string, int>(comparer);
```

Класс `OrderedDictionary` – слаботипизированный класс, запоминающий порядок добавления элементов в словарь. В некотором смысле, данный класс является комбинацией классов `Hashtable` и `ArrayList`. Для доступа к элементам в `OrderedDictionary` можно применять как ключ, так и целочисленный индекс.

Класс `ListDictionary` использует для хранения элементов словаря не хэш-таблицу, а односвязный список. Это делает данный класс неэффективным при работе с большими наборами данных. `ListDictionary` рекомендуется использовать, если количество хранимых элементов не превышает десяти. Класс `HybridDictionary` – это форма словаря, использующая список для хранения при малом количестве элементов, и переключающаяся на применение хэш-функции, когда количество элементов превышает определённый порог. Оба класса `ListDictionary` и `HybridDictionary` являются слаботипизированными и находятся в пространстве имён `System.Collections.Specialized`.

Платформа .NET предоставляет три класса-словаря, организованных так, что их элементы всегда отсортированы по ключу: `SortedDictionary<TKey, TValue>`, `SortedList` и `SortedList<TKey, TValue>`. Данные классы используют разные внутренние структуры для хранения элементов словаря. Класс `SortedDictionary<TKey, TValue>` быстрее классов `SortedList` и `SortedList<TKey, TValue>` при выполнении вставки элементов. Но классы `SortedList` и `SortedList<TKey, TValue>` могут предоставить возможность, которой нет у `SortedDictionary<TKey, TValue>` – доступ к элементу не только по ключу, но и с использованием целочисленного индекса.

14. Типы для создания пользовательских коллекций

Типы коллекций, описанные в предыдущих параграфах, применимы в большинстве стандартных ситуаций. Однако иногда требуется создать собственный тип-коллекцию. Например, в случае, когда изменение коллекции должно генерировать событие, или когда необходима дополнительная проверка данных при помещении их в коллекцию. Для облегчения решения этой задачи платформа .NET предлагает несколько классов, размещённых в пространстве имён `System.Collections.ObjectModel`.

Универсальный класс `Collection<T>` является настраиваемой оболочкой над классом `List<T>`[1]. В дополнение к реализации интерфейсов `IList<T>` и `IList`, класс `Collection<T>` определяет четыре виртуальных метода `ClearItems()`, `InsertItem()`, `RemoveItem()`, `SetItem()` и свойство для чтения `Items`, имеющее тип `IList<T>`. Переопределяя виртуальные методы, можно модифицировать нормальное поведение класса `List<T>` при изменении набора.

Рассмотрим пример использования `Collection<T>`. Пусть класс `Track` представляет отдельную композицию музыкального альбома. Класс `Album` наследуется от `Collection<Track>` и описывает альбом. Виртуальные методы класса `Collection<Track>` переопределяются, чтобы корректно изменять значение свойства `Album` у объекта `Track`.

```
public class Track
{
    public string Title { get; set; }
    public uint Length { get; set; }
    public Album Album { get; internal set; }
}

public class Album : Collection<Track>
{
    protected override void InsertItem(int index, Track item)
    {
        base.InsertItem(index, item);
        item.Album = this;
    }

    protected override void SetItem(int index, Track item)
    {
        base.SetItem(index, item);
        item.Album = this;
    }

    protected override void RemoveItem(int index)
    {
        this[index].Album = null;
```

[1] В пространстве имён `System.Collections` имеется слаботипизированный аналог класса `Collection<T>` – класс `CollectionBase`.

```
            base.RemoveItem(index);
        }

        protected override void ClearItems()
        {
            foreach (Track track in this)
            {
                track.Album = null;
            }
            base.ClearItems();
        }
    }
```

У класса `Collection<T>` имеется конструктор, принимающий в качестве аргумента объект, реализующий `IList<T>`. В отличие от других классов коллекций, этот набор не копируется – запоминается ссылка на него. То есть, изменение набора будет означать изменение коллекции `Collection<T>` (хотя и без вызова виртуальных методов).

Класс `ReadOnlyCollection<T>` – это наследник `Collection<T>`, предоставляющий доступ для чтения элементов, но не для модификации коллекции. Конструктор класса принимает в качестве аргумента объект, реализующий `IList<T>`. Класс не содержит открытых методов добавления или удаления элемента, но можно получить доступ к элементу по индексу и изменить его.

```
var album = new Album
                {
                    new Track {Title = "Speak To Me", Length = 68},
                    new Track {Title = "Breathe", Length = 168},
                    new Track {Title = "On The Run", Length = 230}
                };

var albumReadOnly = new ReadOnlyCollection<Track>(album);
albumReadOnly[1].Title = string.Empty;
```

Класс `ObservableCollection<T>` – это коллекция, позволяющая отслеживать модификации своего набора данных. Этот класс наследуется от `Collection<T>` и реализует интерфейс `INotifyCollectionChanged`, который описывает событие, генерируемое при изменении данных:

```
public interface INotifyCollectionChanged
{
    event NotifyCollectionChangedEventHandler CollectionChanged;
}
```

Аргумент события `CollectionChanged` позволяет узнать, какое действие выполнено над набором данных (добавление, удаление, замена элемента), а также получить информацию о новых или удалённых элементах коллекции.

```
// коллекция album – такая же, как в предыдущем примере
var observable = new ObservableCollection<Track>(album);
observable.CollectionChanged += (sender, e) =>
                                {
                                    Console.WriteLine(e.Action);
                                    foreach (Track item in e.NewItems)
                                    {
                                        Console.WriteLine(item.Title);
                                    }
                                };
observable.Add(new Track {Title = "Time", Length = 424});
```

Абстрактный класс `KeyedCollection<TKey, TItem>` является наследником `Collection<T>`. Этот класс добавляет возможность обращения к элементу по ключу (как в словарях). При использовании `KeyedCollection<TKey, TItem>` требуется переопределить метод `GetKeyForItem()` для вычисления ключа элемента.

Для демонстрации применения `KeyedCollection<TKey, TItem>` модифицируем пример с классами `Track` и `Album`:

```
public class Track
{
    private string _title;

    public string Title
    {
        get { return _title; }
        set
        {
            if (Album != null && _title != value)
            {
                Album.ChangeTitle(this, value);   // изменение ключа
            }
            _title = value;
        }
    }

    public uint Length { get; set; }
    public AlbumDictionary Album { get; internal set; }
}

public class AlbumDictionary : KeyedCollection<string, Track>
{
    protected override string GetKeyForItem(Track item)
    {
        return item.Title;       // ключом будет название композиции
    }

    internal void ChangeTitle(Track track, string title)
    {
```

```
            ChangeItemKey(track, title);   // метод меняет ключ элемента
        }

        // методы ClearItems(), InsertItem(), RemoveItem(), SetItem()
        // реализованы так же, как в классе Album
    }

    // пример использования
    var album = new AlbumDictionary
                    {
                        new Track {Title = "Speak To Me", Length = 68},
                        new Track {Title = "Breathe", Length = 168},
                        new Track {Title = "On The Run", Length = 230}
                    };
    album[0].Length = 0;                    // обращение по индексу
    album["Speak To Me"].Length = 68;       // обращение по ключу
```

15. Технология LINQ to Objects

Платформа .NET версии 3.5 представила новую технологию работы с коллекциями – *Language Integrated Query* (LINQ). По типу обрабатываемой информации LINQ делится на LINQ to Objects – библиотеки для обработки коллекций объектов в памяти, LINQ to SQL – библиотеки для работы с базами данных, LINQ to XML предназначена для обработки XML-информации. В данном параграфе акцент сделан на LINQ to Objects.

Технически, LINQ to Objects – это набор классов, содержащих типичные методы обработки коллекций: поиск данных, сортировка, фильтрация. Ядром LINQ to Objects является статический класс `Enumerable`, размещённый в пространстве имён `System.Linq`[1] . Этот класс содержит набор методов расширения интерфейса `IEnumerable<T>`, которые в дальнейшем будут называться *операторами LINQ*. Для удобства дальнейшего изложения используем стандартное деление операторов LINQ на группы в зависимости от выполняемых действий:

1. Оператор условия `Where` (отложенные вычисления).
2. Операторы проекций (отложенные вычисления).
3. Операторы упорядочивания (отложенные вычисления).
4. Оператор группировки `GroupBy` (отложенные вычисления).
5. Операторы соединения (отложенные вычисления).
6. Операторы работы с множествами (отложенные вычисления).
7. Операторы агрегирования.
8. Операторы генерирования (отложенные вычисления).
9. Операторы кванторов и сравнения.
10. Операторы разбиения (отложенные вычисления).
11. Операторы элемента.
12. Операторы преобразования.

[1] Для использования `System.Linq` необходимо подключить сборку `System.Core.dll`.

В примерах параграфа будут использоваться либо коллекции примитивных типов, либо коллекция `gr` объектов класса `Student`:

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
    public IEnumerable<int> Marks { get; set; }
}

var gr = new List<Student>{
new Student {Name = "Smirnov", Age = 18, Marks = new[] {10, 8, 9}},
new Student {Name = "Ivanova", Age = 20, Marks = new[] {5, 6, 9}},
new Student {Name = "Kuznetsov", Age = 18, Marks = new[] {7, 7, 4}},
new Student {Name = "Sokolov", Age = 20, Marks = new[] {7, 8, 8}},
new Student {Name = "Lebedeva", Age = 20, Marks = new[] {9, 9, 9}}
};
```

1. Оператор условия Where().

Оператор производит фильтрацию коллекции, основываясь на аргументе-предикате. Сигнатуры оператора[1]:

```
IEnumerable<T> Where<T>(this IEnumerable<T> source,
                        Func<T, bool> predicate);

IEnumerable<T> Where<T>(this IEnumerable<T> source,
                        Func<T, int, bool> predicate);
```

Второй вариант оператора `Where()` позволяет передать аргументу-предикату индекс элемента в коллекции (заметим, что многие другие операторы имеют перегруженную версию, устроенную по такому же принципу).

Примеры использования `Where()`:

```
var list = new List<int> {1, 3, -1, -4, 7};
var r1 = list.Where(x => x < 0);
var r2 = gr.Where(student => student.Age > 19);
var r3 = gr.Where((student, pos) => student.Age > 19 && pos < 3);
```

2. Операторы проекций.

Операторы проекций применяются для выборки информации, при этом они могут изменять тип элементов итоговой коллекции. Основным оператором проекции является `Select()`:

```
IEnumerable<S> Select<T, S>(this IEnumerable<T> source,
                            Func<T, S> selector);
```

[1] Все операторы LINQ имеют модификаторы `public static`. Для краткости эти модификаторы не указываются.

Оператор SelectMany() может применяться, если результатом проекции является набор данных. В этом случае оператор соединяет все элементы набора в одну коллекцию.

```
IEnumerable<S> SelectMany<T, S>(this IEnumerable<T>source,
                                Func<T, IEnumerable<S>> selector);
```

Примеры использования операторов проекций:

```
var r1 = gr.Select(student => student.Name);
var r2 = gr.Select(student => new {student.Name, student.Age});
var r3 = gr.SelectMany(student => student.Marks);
```

Коллекция r1 будет содержать имена студентов. Коллекция r2 состоит из объектов анонимного типа с полями Name и Age. Коллекция r3 – это все оценки студентов (15 элементов типа int).

3. Операторы упорядочивания.

Операторы OrderBy() и OrderByDescending() выполняют сортировку коллекции по возрастанию или убыванию соответственно. Имеется версия данных операторов, принимающая в качестве дополнительного аргумента объект, реализующий IComparer<T>.

```
IOrderedEnumerable<T> OrderBy<T, K>(this IEnumerable<T> source,
                                    Func<T, K> keySelector);

IOrderedEnumerable<T> OrderByDescending<T, K>(
                this IEnumerable<T> source, Func<T, K> keySelector);
```

Интерфейс IOrderedEnumerable<T> является наследником IEnumerable<T> и описывает упорядоченную последовательность элементов с указанием на ключ сортировки. Если после выполнения сортировки по одному ключу требуется дополнительная сортировка по другому ключу, нужно воспользоваться операторами ThenBy() и ThenByDescending(). Имеется также оператор Reverse(), обращающий коллекцию.

Пример использования операторов упорядочивания:

```
var r1 = Enumerable.Reverse(gr);

var r2 = gr.OrderBy(student => student.Age);

var r3 = gr.OrderByDescending(student => student.Age)
          .ThenBy(student => student.Name);
```

Чтобы получить коллекцию r1, метод расширения использовался как обычный статический метод класса, так как у List<T> имеется собственный метод Reverse(). В коллекции r3 студенты упорядочены по убыванию возраста, а при совпадении возрастов – по фамилиям в алфавитном порядке.

4. Оператор группировки GroupBy().

Оператор группировки `GroupBy()` разбивает коллекцию на группы элементов с одинаковым значением некоторого ключа. Оператор `GroupBy()` имеет перегруженные версии, позволяющие указать селектор ключа, преобразователь элементов в группе, объект, реализующий `IEqualityComparer<T>` для сравнения ключей. Простейшая версия оператора группировки имеет следующий вид:

```
IEnumerable<IGrouping<K, T>> GroupBy<T, K>(this IEnumerable<T> src,
                                           Func<T, K> keySelector);
```

Здесь последний параметр указывает на функцию, которая строит по элементу поле-ключ. Обычно эта функция просто выбирает одно из полей объекта. Интерфейс `IGrouping<K, T>` унаследован от `IEnumerable<T>` и содержит дополнительное типизированное свойство `Key` – ключ группировки.

Рассмотрим применение оператора группировки. Сгруппируем студентов по возрасту и для каждой группы возрастов выведем ключ и элементы:

```
var r1 = gr.GroupBy(student => student.Age);
foreach (IGrouping<int, Student> group in r1)
{
    Console.WriteLine(group.Key);
    foreach (Student student in group)
    {
        Console.WriteLine(student.Name);
    }
}
```

5. Операторы соединения.

Операторы соединения применяются, когда требуется соединить две коллекции, элементы которых имеют общие атрибуты. Основным оператором соединения является оператор `Join()`.

```
IEnumerable<V> Join<T, U, K, V>(this IEnumerable<T> outer,
                                IEnumerable<U> inner,
                                Func<T, K> outerKeySelector,
                                Func<U, K> innerKeySelector,
                                Func<T, U, V> resultSelector);
```

Оператор `Join()` требует задания двух коллекций (внешней и внутренней) и трёх функций. Первая функция порождает ключ из элемента внешней коллекции, вторая – из элемента внутренней коллекции, а третья функция продуцирует объект коллекции-результата. При выполнении соединения `Join()` итерируется по внешней коллекции и ищет соответствия с элементами внутренней коллекции. При этом возможны следующие ситуации:

1. Найдено одно соответствие – в результат включается один элемент.
2. Найдено множественное соответствие – результат содержит по элементу для каждого соответствия.

3. Соответствий не найдено – элемент не входит в результат.

Рассмотрим примеры использования оператора `Join()`. Для этого опишем коллекцию `cit` объектов класса `Citizen`.

```
public class Citizen
{
    public int BirthYear { get; set; }
    public string IDNumber { get; set; }
}

int year = DateTime.Now.Year;
var cit = new List<Citizen> {
          new Citizen {BirthYear = year - 17, IDNumber = "KLM897"},
          new Citizen {BirthYear = year - 18, IDNumber = "WEF442"},
          new Citizen {BirthYear = year - 18, IDNumber = "HHH888"},
          new Citizen {BirthYear = year - 25, IDNumber = "XYZ012"}};
```

Выполним оператор `Join()`:

```
var r1 = gr.Join(cit,
            student => year - student.Age,
            citizen => citizen.BirthYear,
 (student, citizen) => new {student.Name, citizen.IDNumber});

// r1 содержит следующие объекты:
// {Name = "Smirnov", IDNumber = "WEF442"}
// {Name = "Smirnov", IDNumber = "HHH888"}
// {Name = "Kuznetsov", IDNumber = "WEF442"}
// {Name = "Kuznetsov", IDNumber = "HHH888"}
```

Оператор `GroupJoin()` порождает набор, группируя элементы внутренней коллекции при нахождении соответствия с элементом внешней коллекции: Если же соответствие не найдено, в результат включается пустая группа.

```
IEnumerable<V> GroupJoin()<T, U, K, V>(this IEnumerable<T> outer,
                                       IEnumerable<U> inner,
                                       Func<T, K> outerKeySelector,
                                       Func<U, K> innerKeySelector,
                            Func<T, IEnumerable<U>, V> resultSelector);
```

Ниже приведён пример использования `GroupJoin()`.

```
var r3 = cit.GroupJoin(gr,
           citizen => citizen.BirthYear,
           student => year - student.Age,
  (citizen, group) => new {citizen.IDNumber,
                           Names = group.Select(st => st.Name)});

foreach (var data in r3)
{
```

```
        Console.WriteLine(data.IDNumber);
        foreach (string s in data.Names)
        {
            // 1-я и 4-я группы пусты,
            // 2-я и 3-я содержат по два элемента
            Console.WriteLine(s);
        }
    }
```

Оператор `Zip()` порождает набор на основе двух исходных коллекций, выполняя заданное генерирование элементов. Длина результирующей коллекции равна длине меньшей из двух исходных коллекций.

```
IEnumerable<V> Zip<T, U, V>(this IEnumerable<T> first,
                            IEnumerable<U> second,
                            Func<T, U, V> resultSelector);
```

Дадим простейший пример использования `Zip()`:

```
int[] numbers = {1, 2, 3, 4};
string[] words = {"one", "two", "three"};
var res = numbers.Zip(words, (f, s) => f + "=" + s);   // 3 элемента
```

6. Операторы работы с множествами.

В LINQ to Objects имеется набор операторов для работы с множествами.

```
IEnumerable<T> Distinct<T>(this IEnumerable<T> source);

IEnumerable<T> Union<T>(this IEnumerable<T> first,
                        IEnumerable<T> second);

IEnumerable<T> Intersect<T>(this IEnumerable<T> first,
                            IEnumerable<T> second);

IEnumerable<T> Except<T>(this IEnumerable<T> first,
                         IEnumerable<T> second);
```

Оператор `Distinct()` удаляет из коллекции повторяющиеся элементы. Операторы `Union()`, `Intersect()` и `Except()` представляют объединение, пересечение и разность двух множеств.

7. Операторы агрегирования.

К операторам агрегирования относятся операторы, результатом работы которых является скалярное значение. Следующие операторы возвращают количество элементов коллекции. При этом может быть использована перегруженная версия, принимающая в качестве второго аргумента предикат фильтрации.

```
int Count<T>(this IEnumerable<T> source);

long LongCount<T>(this IEnumerable<T> source);
```

Следующие операторы подсчитывают сумму и среднее значение в коллекции. При этом *Num* должен быть типом `int`, `long`, `float`, `double`, `decimal` или вариантом этих типов с поддержкой `null` (например, `long`?).

```
Num Sum(this IEnumerable<Num> source);

Num Sum<T>(this IEnumerable<T> source, Func<T, Num> selector);

Num Average(this IEnumerable<Num> source);

Num Average<T>(this IEnumerable<T> source, Func<T, Num> selector);
```

Существует также несколько перегруженных версий операторов для нахождения минимального и максимального значений. Первые две версии применяются для коллекций с числовым элементом. Последние две предполагают, что элемент коллекции реализует интерфейс `IComparable<T>`.

```
Num Min/Max(this IEnumerable<Num> source);

Num Min<T>/Max<T>(this IEnumerable<T> src, Func<T, Num> selector);

T Min<T>/Max<T>(this IEnumerable<T> source);

S Min<T, S>/Max<T, S>(this IEnumerable<T> src, Func<T, S> selector);
```

Оператор `Aggregate()` позволяет выполнить для коллекции собственный алгоритм агрегирования. Его простейшая форма:

```
T Aggregate<T>(this IEnumerable<T> source, Func<T, T, T> func);
```

Функция `func` принимает два аргумента: значение-аккумулятор и текущее значение коллекции. Результат функции перезаписывается в аккумулятор. В следующем примере оператор `Aggregate()` применяется для обращения строки:

```
var text = "The quick brown fox jumps over the lazy dog";
string[] words = text.Split(' ');
var reversed = words.Aggregate((acc, next) => next + " " + acc);
```

8. Операторы генерирования.

Эта группа операторов позволяет создать набор данных. Первый оператор группы – оператор `Range()`. Он просто выдаёт указанное количество подряд идущих целых чисел, начиная с заданного значения.

```
IEnumerable<int> Range(int start, int count);
```

Продемонстрируем использование `Range()` в задаче поиска простых чисел, которую решим при помощи LINQ:

```
var primes = Enumerable.Range(2, 999)
             .Where(x => !Enumerable.Range(2, (int) Math.Sqrt(x))
                                    .Any(y => x != y && x%y == 0));
foreach (var prime in primes)
    Console.WriteLine(prime);
```

Следующий оператор генерирования – оператор `Repeat()`. Он создаёт коллекцию, в которой указанный элемент повторяется требуемое число раз. Для ссылочных типов дублируются ссылки, а не содержимое.

```
IEnumerable<T> Repeat<T>(T element, int count);
```

Покажем не совсем стандартное применение `Repeat()` для генерирования последовательности случайных чисел:

```
Random rnd = new Random();
var r1 = Enumerable.Repeat(0, 20).Select(i => rnd.Next(0, 40));
```

Последним генерирующим оператором является оператор `Empty()`, который порождает пустое перечисление определённого типа.

```
IEnumerable<T> Empty<T>();
```

9. Операторы кванторов и сравнения.

Операторы кванторов похожи на соответствующие операторы в математической логике.

```
bool Any<T>(this IEnumerable<T> source, Func<T, bool> predicate);

bool Any<T>(this IEnumerable<T> source);

bool All<T>(this IEnumerable<T> source, Func<T, bool> predicate);

bool Contains<T>(this IEnumerable<T> source, T value);

bool Contains<T>(this IEnumerable<T> source, T value,
                 IEqualityComparer<T> comparer)
```

Оператор `Any()` проверяет наличие хотя бы одного элемента в коллекции, удовлетворяющего указанному предикату. Вторая версия оператора `Any()` просто проверяет коллекцию на непустоту. Оператор `All()` возвращает `true`, если все элементы коллекции удовлетворяют предикату. И, наконец, оператор `Contains()` проверяет, входит ли заданное значение в коллекцию.

Оператор `SequenceEqual()` сравнивает две коллекции и возвращает `true`, если обе коллекции имеют одну длину, и их соответствующие элементы равны:

```
bool SequenceEqual<T>(this IEnumerable<T> first,
                      IEnumerable<T> second);
```

```
bool SequenceEqual<T>(this IEnumerable<T> first,
                      IEnumerable<T> second,
                      IEqualityComparer<T> comparer);
```

10. Операторы разбиения.

Операторы разбиения выделяют некую часть исходной коллекции (например, первые десять элементов).

```
IEnumerable<T> Take<T>(this IEnumerable<T> source, int count);


IEnumerable<T> TakeWhile<T>(this IEnumerable<T> source,
                            Func<T, bool> predicate);

IEnumerable<T> Skip<T>(this IEnumerable<T> source, int count);

IEnumerable<T> SkipWhile<T>(this IEnumerable<T> source,
                            Func<T, bool> predicate);
```

Комбинация операторов `Take()` и `Skip()` часто применяется, чтобы организовать постраничный просмотр информации (например, просмотр записей из большой таблицы):

```
var bigTable = Enumerable.Range(1, 1000);
int pageSize = 20, pageNumber = 6;
var page = bigTable.Skip((pageNumber - 1)*pageSize).Take(pageSize);
```

11. Операторы элемента.

Эта группа операторов предназначена для выделения из коллекции единственного элемента, удовлетворяющего определённым условиям.

Оператор `First()` выделяет первый элемент (или первый элемент, удовлетворяющий определённому предикату).

```
T First<T>(this IEnumerable<T> source);

T First<T>(this IEnumerable<T> source, Func<T, bool> predicate);
```

Если в коллекции нет элементов, или не нашлось элементов, удовлетворяющих предикату, оператор `First()` выбрасывает исключение `InvalidOperationException`. Если требуется, чтобы исключение не выбрасывалось, а возвращалось предопределённое значение для типа, следует использовать оператор `FirstOrDefault()`.

```
T FirstOrDefault<T>(this IEnumerable<T> source);

T FirstOrDefault<T>(this IEnumerable<T> source,
                    Func<T, bool> predicate);
```

Аналогично работают операторы `Last()` и `LastOrDefault()` для выделения последнего элемента.

Операторы `Single()` и `SingleOrDefault()` рассчитаны на то, что коллекция (или набор элементов, удовлетворяющих предикату) будет состоять из одного элемента, который данные операторы и возвращают. Если в коллекции нет элементов, или их оказалось больше одного, оператор `Single()` выбрасывает исключение `InvalidOperationException`. Оператор `SingleOrDefault()` выбрасывает такое исключение, если элементов оказалось больше одного.

Пара операторов `ElementAt()` и `ElementAtOrDefault()` пытаются вернуть элемент на указанной целочисленной позиции.

Оператор `DefaultIfEmpty()` проверяет коллекцию на пустоту. Если в коллекции нет элементов, то возвращается или значение по умолчанию для типа, или указанное значение. Если коллекция непустая, то она и возвращается.

```
IEnumerable<T> DefaultIfEmpty<T>(this IEnumerable<T> source);

IEnumerable<T> DefaultIfEmpty<T>(this IEnumerable<T> source,
                                 T defaultValue);
```

12. Операторы преобразования.

Назначение данных операторов – преобразования универсальных коллекций, реализующих `IEnumerable<T>`, в конкретные типы.

Оператор `AsEnumerable()` возвращает свой аргумент, приведённый к типу `IEnumerable<T>`. Этот оператор необходим в том случае, если тип аргумента `source` имеет методы, аналогичные имеющимся в LINQ to Objects.

```
IEnumerable<T> AsEnumerable<T>(this IEnumerable<T> source);
```

Операторы `ToArray()` и `ToList()` выполняют преобразование коллекции в массив или список.

```
T[] ToArray<T>(this IEnumerable<T> source);

List<T> ToList<T>(this IEnumerable<T> source);
```

Существует несколько версий оператора `ToDictionary()`, порождающего из коллекции словарь. Например, следующая версия использует отдельные функции для выделения из элемента коллекции ключа и значения ключа.

```
Dictionary<K, V> ToDictionary<T, K, V>(this IEnumerable<T> source,
                                       Func<T, K> keySelector,
                                       Func<T, V> valueSelector);
```

Оператор `ToLookup()` преобразовывает коллекцию в объект класса `Lookup<K, V>` из пространства имён `System.Linq`. Класс `Lookup<K, V>` представляет словарь, в котором ключу сопоставлен не единственный элемент, а набор элементов.

```
ILookup<K, V> ToLookup<T, K, V>(this IEnumerable<T> source,
                                Func<T, K> keySelector,
                                Func<T, V> valueSelector);
```

Оператор `OfType<T>()` итерируется по коллекции и генерирует список, содержащий только элементы заданного типа `T`.

```
IEnumerable<T> OfType<T>(this IEnumerable source);
```

Оператор `Cast<T>()` итерируется по слаботипизированной коллекции и пытается выполнить приведение каждого элемента к указанному типу. Если приведение выполнить не удаётся, генерируется исключение. Данный оператор полезен, если мы хотим применять LINQ для старых коллекций, таких, как `ArrayList`.

```
IEnumerable<T> Cast<T>(this IEnumerable source);
```

Из операторов преобразования отложенные вычисления выполняют операторы `AsEnumerable<T>()`, `OfType<T>()`, `Cast<T>()`.

Язык C# версии 3.0 вводит новые ключевые слова и особые синтаксические расширения для записи некоторых операторов LINQ в виде выражений запросов. Составленное выражение запросов должно подчиняться следующим правилам (за строгим описанием правил следует обратиться к MSDN):

1. Выражение должно начинаться с конструкции `from`, которая указывает на обрабатываемую коллекцию.

2. Затем выражение может содержать ноль или более конструкций `from`, `let` или `where`. Конструкция `let` представляет переменную и присваивает ей значение. Конструкция `where` фильтрует элементы коллекции.

3. Затем выражение может включать ноль или более конструкций `orderby`, с полями сортировки и необязательным указанием на направление упорядочивания. Направление может быть `ascending` или `descending`.

4. Затем следует конструкция `select` или `group`.

5. Наконец, в оставшейся части выражения может следовать необязательная конструкция продолжения. Такой конструкцией является `into`.

Выражения запросов транслируются компилятором в вызовы соответствующих операторов LINQ. Приведём некоторые примеры эквивалентной записи запросов к коллекциям данных.

```
var r1 = gr.Where(s => s.Age > 20).Select(s => s.Marks);
// эквивалентный синтаксис
var r2 = from s in gr
         where s.Age > 20
         select s.Marks;
var r3 = gr.OrderByDescending(s => s.Age).ThenBy(s => s.Name);
// эквивалентный синтаксис
var r4 = from s in gr
         orderby s.Age descending, s.Name
         select s;
```

```
var r5 = gr.Select(student => new {student.Name, student.Age});
// эквивалентный синтаксис
var r6 = from student in gr select new {student.Name, student.Age};
```

Приведём несколько примеров использования технологии LINQ to Objects. В первом примере подсчитывается частота вхождения каждого слова в текст.

```
var text = "this is the text and this is the code for the text";
var words = text.Split(' ');
var result = words.GroupBy(word => word.ToUpper())
            .Select(group => new {group.Key, Count = group.Count()})
            .OrderBy(pair => pair.Count);

foreach (var pair in result)
    Console.WriteLine("{0} - {1}", pair.Count, pair.Key);
```

Второй пример – метод расширения, демонстрирующий идею алгоритма быстрой сортировки (так как LINQ to Objects использует отложенные вычисления, метод выполняется медленно).

```
public static IEnumerable<T> QSort<T>(this IEnumerable<T> data)
                                      where T : IComparable<T>
{
    var tail = data.Skip(1);
    if (tail.Any())
    {
        var elem = data.First();
        var head = data.Take(1);
        var left = tail.Where(x => x.CompareTo(elem) < 0).QSort();
        var right = tail.Where(x => x.CompareTo(elem) >= 0).QSort();
        return left.Concat(head).Concat(right);
    }
    return data;
}
```

16. Работа с объектами файловой системы

В пространстве имён `System.IO` доступно несколько классов для работы с объектами файловой системы – дисками, каталогами, файлами.

Класс `DriveInfo` инкапсулирует информацию о диске. Он имеет статический метод `GetDrives()` для получения массива объектов `DriveInfo`, соответствующих дискам операционной системы. В примере кода демонстрируется работа с элементами класса `DriveInfo`.

```
var allDrives = DriveInfo.GetDrives();
foreach (var d in allDrives)
{
    Console.WriteLine("Drive name: {0}", d.Name);
    Console.WriteLine("Drive type: {0}", d.DriveType);
    if (!d.IsReady) continue;
```

```
        Console.WriteLine("Volume Label: {0}", d.VolumeLabel);
        Console.WriteLine("File system: {0}", d.DriveFormat);
        Console.WriteLine("Root: {0}", d.RootDirectory);
        Console.WriteLine("Total size: {0}", d.TotalSize);
        Console.WriteLine("Free size: {0}", d.TotalFreeSpace);
        Console.WriteLine("Available: {0}", d.AvailableFreeSpace);
    }
```

Классы `Directory`, `File`, `DirectoryInfo` и `FileInfo` предназначены для работы с каталогами и файлами. Первые два класса выполняют операции при помощи статических методов, вторые два – при помощи экземплярных методов.

Рассмотрим работу с классами `DirectoryInfo` и `FileInfo`. Данные классы являются наследниками абстрактного класса `FileSystemInfo`. Этот класс содержит следующие основные элементы, перечисленные в табл. 11.

Таблица 11

Элементы класса `FileSystemInfo`

Имя элемента	Описание
`Attributes`	Свойство позволяет получить или установить атрибуты объекта файловой системы (тип – перечисление `FileAttributes`)
`CreationTime`	Время создания объекта файловой системы
`Exists`	Свойство для чтения, проверка существования объекта файловой системы
`Extension`	Свойство для чтения, расширение файла
`FullName`	Свойство для чтения, полное имя объекта файловой системы
`LastAccessTime`, `LastAccessTimeUtc`	Время последнего доступа к объекту файловой системы (локальное или всемирное координированное)
`LastWriteTime`, `LastWriteTimeUtc`	Время последней записи для объекта файловой системы (локальное или всемирное координированное)
`Name`	Свойство для чтения, имя файла или каталога
`Delete()`	Метод удаляет объект файловой системы
`Refresh()`	Метод обновляет информацию об объекте файловой системы

Конструктор класса `DirectoryInfo` принимает в качестве аргумента строку с именем того каталога, с которым будет производиться работа. Для указания текущего каталога используется строка `"."`. При попытке работать с данными несуществующего каталога генерируется исключение. Работа с методами и свойствами класса `DirectoryInfo` показана в следующем фрагменте кода:

```
var dir = new DirectoryInfo(@"C:\Temp");

// выводим некоторые свойства каталога C:\Temp
Console.WriteLine("Full name: {0}", dir.FullName);
Console.WriteLine("Parent directory: {0}", dir.Parent);
Console.WriteLine("Root directory: {0}", dir.Root);
Console.WriteLine("Creation date: {0}", dir.CreationTime);

// создаём подкаталог
dir.CreateSubdirectory("Subdir");
```

Класс `DirectoryInfo` обладает двумя наборами методов для получения дочерних подкаталогов, файлов или объектов `FileSystemInfo`. Методы вида `Get`*`Элементы`*`()` выполняются немедленно и возвращают массив. Методы вида `Enumerate`*`Элементы`*`()` используют отложенное выполнение и возвращают перечислитель:

```
var dir = new DirectoryInfo(@"C:\Temp");

// получаем файлы по маске из всех подкаталогов
FileInfo[] f = dir.GetFiles("*.txt", SearchOption.AllDirectories);

// получаем файлы, используя отложенное выполнение
foreach (var fileInfo in dir.EnumerateFiles())
    Console.WriteLine(fileInfo.Name);
```

Класс `FileInfo` описывает файл на диске и позволяет производить операции с этим файлом. Наиболее важные элементы класса представлены в табл. 12.

Таблица 12

Элементы класса `FileInfo`

Имя элемента	Описание
`AppendText()`	Создаёт объект `StreamWriter` для добавления текста к файлу
`CopyTo()`	Копирует существующий файл в новый файл
`Create()`	Создаёт файл и возвращает объект `FileStream` для работы
`CreateText()`	Создаёт объект `StreamWriter` для записи текста в новый файл
`Decrypt()`	Дешифрует файл, зашифрованный методом `Encrypt()`
`Directory`	Свойство для чтения, каталог файла
`DirectoryName`	Свойство для чтения, полный путь к файлу
`Encrypt()`	Шифрует файл с учётом системных данных текущего пользователя
`IsReadOnly`	Булево свойство. Указывает, является ли файл файлом только для чтения
`Length`	Свойство для чтения, размер файла в байтах
`MoveTo()`	Перемещает файл (возможно, с переименованием)
`Open()`	Открывает файл с указанными правами доступа
`OpenRead()`	Создаёт объект `FileStream`, доступный только для чтения
`OpenText()`	Создаёт объект `StreamReader` для чтения информации из текстового файла
`OpenWrite()`	Создаёт объект `FileStream`, доступный для чтения и записи

Как правило, код, работающий с данными файла, вначале вызывает метод `Open()`. Рассмотрим перегруженную версию метода `Open()` с тремя параметрами. Первый параметр определяет режим запроса на открытие файла. Для него используются значения из перечисления `FileMode`:

`Append` – открывает файл, если он существует, и ищет конец файла. Если файл не существует, то он создаётся. Этот режим может использоваться только с доступом `FileAccess.Write`;

`Create` – указывает на создание нового файла. Если файл существует, он будет перезаписан;

`CreateNew` – указывает на создание нового файла. Если файл существует, генерирует исключение `IOException`;

`Open` – операционная система должна открыть существующий файл;

`OpenOrCreate` – операционная система должна открыть существующий файл или создать новый, если файл не существует;

`Truncate` – система должна открыть существующий файл и обрезать его до нулевой длины.

Рис. 4 показывает выбор `FileMode` в зависимости от задачи.

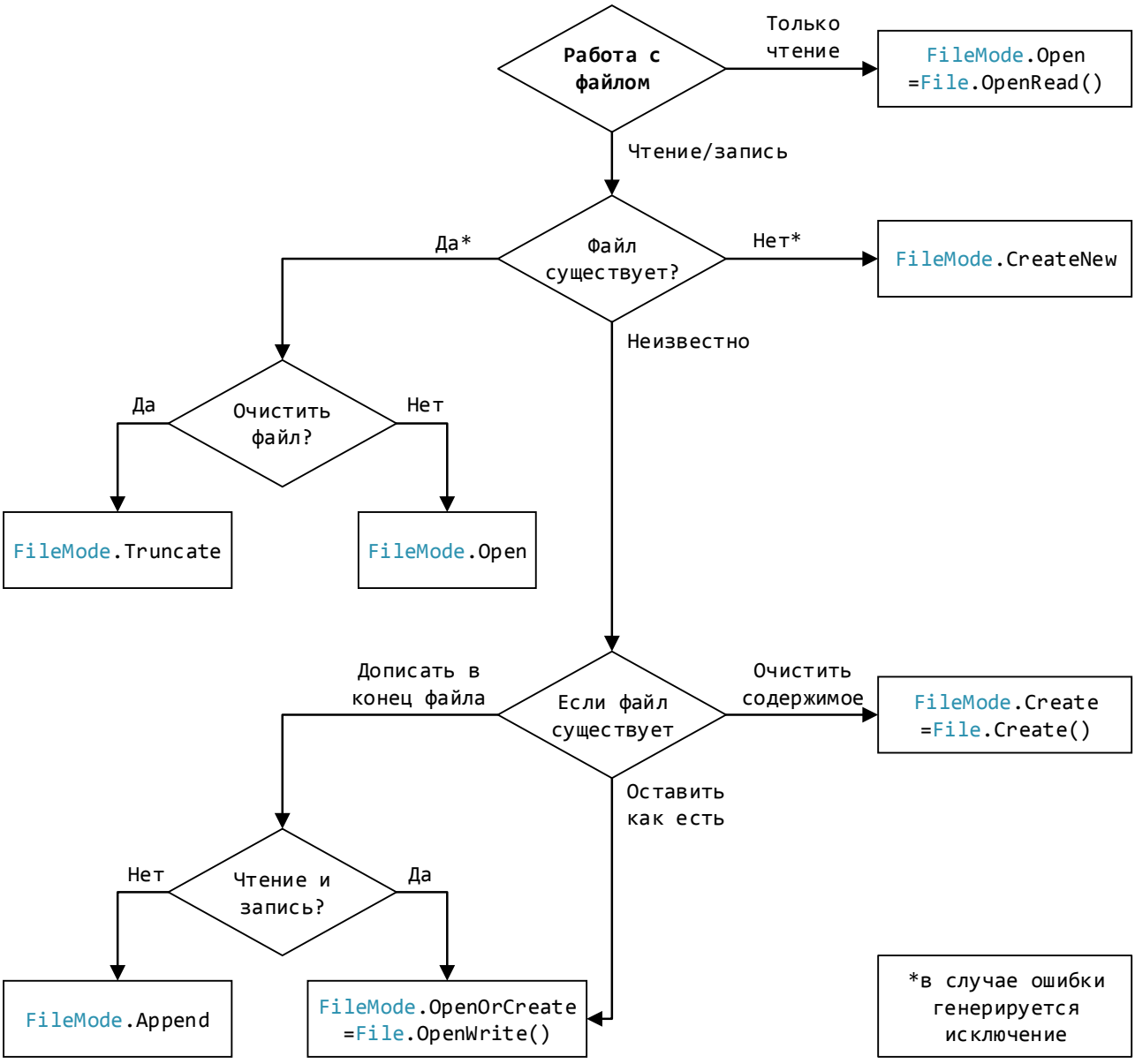

Рис. 4. Выбор значения `FileMode`.

Второй параметр метода `Open()` определяет тип доступа к данным файла. Для него используются элементы перечисления `FileAccess`:

`Read` – файл будет открыт только для чтения;

`ReadWrite` – файл будет открыт и для чтения, и для записи;

`Write` – файл открывается только для записи, то есть добавления данных.

Третий параметр задаёт возможность совместной работы с открытым файлом и представлен значениями перечисления `FileShare`:

`None` – совместное использование запрещено, на любой запрос на открытие файла будет возвращено сообщение об ошибке;

`Read` – файл могут открыть и другие пользователи, но только для чтения;

`ReadWrite` – другие пользователи могут открыть файл и для чтения, и для записи;

`Write` – файл может быть открыт другими пользователями для записи.

Вот пример кода, использующего метод `Open()`:

```
var file = new FileInfo(@"C:\Test.txt");
FileStream fs = file.Open(FileMode.OpenOrCreate,
                          FileAccess.ReadWrite, FileShare.None);
```

Кроме методов класса `FileInfo`, статический класс `File` обладает методами, позволяющими легко прочитать и записать информацию, содержащуюся в файле определённого типа:

`File.AppendAllLines()` – добавляет к текстовому файлу набор строк;

`File.AppendAllText()` – добавляет строку к текстовому файлу;

`File.ReadAllBytes()` – возвращает содержимое файла как массив байтов;

`File.ReadAllLines()` – читает текстовый файл как массив строк;

`File.ReadLines()` – читает файл как коллекцию строк, используя отложенные вычисления;

`File.ReadAllText()` – читает содержимое текстового файла как строку;

`File.WriteAllBytes()` – записывает в файл массив байтов;

`File.WriteAllLines()` – записывает в файл массив или коллекцию строк;

`File.WriteAllText()` – записывает текстовый файл как одну строку;

Классы `ZipArchive` и `ZipFile` из пространства имён `System.IO.Compression`[1] обеспечивают работу с ZIP-архивами. Статические методы класса `ZipFile` позволяют создать архив из файлов указанного каталога, извлечь файлы из архива и открыть архив, вернув объект класса `ZipArchive`.

```
// создаём архив
// (можно дополнительно указать степень сжатия)
ZipFile.CreateFromDirectory(@"C:\Temp\Data", @"C:\MyArchive.zip");

// разархивируем
ZipFile.ExtractToDirectory(@"C:\MyArchive.zip", @"C:\NewFolder");
```

Класс `ZipArchive` представляет пакет сжатых файлов и ориентирован на использование совместно с потоками данных. Отдельный файл архива описывается объектом класса `ZipArchiveEntry`.

---

[1] Чтобы использовать указанные классы, подключите сборки `System.IO.Compression.dll` и `System.IO.Compression.FileSystem.dll`.

```
using (ZipArchive zip = ZipFile.OpenRead(@"C:\MyArchive.zip"))
{
    foreach (ZipArchiveEntry entry in zip.Entries)
    {
        Console.WriteLine(entry.FullName);

        // предполагается некая работа с потоком данных
        Stream stream = entry.Open();
    }
}
```

Статический класс `Path` предназначен для работы с именами файлов и путями в файловой системе. Методы этого класса позволяют выделить имя файла из полного пути, скомбинировать для получения пути имя файла и имя каталога. Также класс `Path` обладает методами, генерирующими имя для временного файла или каталога. Для поиска стандартных папок (например, `My Documents`) следует применять метод `GetFolderPath()` класса `System.Environment`, указав в качестве аргумента одно из значений перечисления `Environment.SpecialFolder`.

```
string tempFile = Path.GetTempFileName();      // c:\Temp\tmp144A.tmp
string ext = Path.GetExtension("info.txt");    // .txt
string win =
       Environment.GetFolderPath(Environment.SpecialFolder.Windows);
```

Класс `FileSystemWatcher` позволяет производить мониторинг активности выбранного каталога. У этого класса определены события, которые генерируются, когда файлы или подкаталоги создаются, удаляются, модифицируются, или изменяются их атрибуты. Применение класса `FileSystemWatcher` демонстрирует следующий фрагмент кода:

```
// создаём и настраиваем объект FileSystemWatcher
var watcher = new FileSystemWatcher
                  {
                      Path = @"C:\Temp",
                      Filter = "*.txt",
                      IncludeSubdirectories = true
                  };

// устанавливаем обработчики событий
FileSystemEventHandler handler = (o, e) =>
Console.WriteLine("File {0} was {1}", e.FullPath, e.ChangeType);
watcher.Created += handler;
watcher.Changed += handler;
watcher.Deleted += handler;
watcher.Renamed += (o, e) =>
Console.WriteLine("Renamed: {0} -> {1}", e.OldFullPath, e.FullPath);
watcher.Error += (o, e) =>
Console.WriteLine("Error: {0}", e.GetException().Message);
```

```
// запускаем мониторинг
watcher.EnableRaisingEvents = true;
Console.WriteLine("Monitoring is on. Press <enter> to exit");
Console.ReadLine();

// останавливаем мониторинг, уничтожаем FileSystemWatcher
watcher.Dispose();
```

17. Ввод и вывод информации

Платформа .NET содержит развитый набор типов для операций ввода и вывода информации. Типы для поддержки ввода/вывода можно разбить на две категории: типы для представления потоков данных и адаптеры потоков. *Поток данных* – это абстрактное представление данных в виде последовательности байтов. Поток либо ассоциируется с неким физическим хранилищем (файлами на диске, памятью, сетью), либо декорирует (обрамляет) другой поток, преобразуя данные тем или иным образом. *Адаптеры потоков* служат оболочкой потока, преобразуя информацию определённого формата в набор байтов (сами адаптеры потоками не являются).

17.1. Потоки данных и декораторы потоков

Представим наиболее часто используемые классы для работы с потоками в виде следующих категорий.

1. Абстрактный класс `System.IO.Stream`**.** Это базовый класс для других классов, представляющих потоки.

2. Классы для работы с потоками, связанными с хранилищами.

`FileStream` – класс для работы с файлами, как с потоками (пространство имён `System.IO`).

`MemoryStream` – класс для представления потока в памяти (пространство имён `System.IO`).

`NetworkStream` – работа с сокетами, как с потоками (пространство имён `System.Net.Sockets`).

`PipeStream` – абстрактный класс из пространства имён `System.IO.Pipes`, базовый для классов-потоков, которые позволяют передавать данные между процессами операционной системы.

3. Декораторы потоков.

`DeflateStream` и `GZipStream` – классы для потоков со сжатием данных (пространство имён `System.IO.Compression`).

`CryptoStream` – поток зашифрованных данных (пространство имён `System.Security.Cryptography`).

`BufferedStream` – поток с поддержкой буферизации данных (пространство имён `System.IO`).

4. Адаптеры потоков.

`BinaryReader` и `BinaryWriter` – классы для ввода и вывода примитивных типов в двоичном формате.

`StreamReader` и `StreamWriter` – классы для ввода и вывода информации в строковом представлении.

`XmlReader` и `XmlWriter` – абстрактные классы для ввода/вывода XML.

Элементы абстрактного класса `Stream` сведены в табл. 13.

Таблица 13

Элементы абстрактного класса `Stream`

Категория	Элементы
Чтение данных	`bool CanRead { get; }`
	`IAsyncResult BeginRead(byte[] buffer, int offset, int count, AsyncCallback callback, object state)`
	`int EndRead(IAsyncResult asyncResult)`
	`int Read(byte[] buffer, int offset, int count)`
	`Task<int> ReadAsync(byte[] buffer, int offset, int count)`
	`int ReadByte()`
Запись данных	`bool CanWrite { get; }`
	`IAsyncResult BeginWrite(byte[] buffer, int offset, int count, AsyncCallback callback, object state)`
	`int EndWrite(IAsyncResult asyncResult)`
	`void Write(byte[] buffer, int offset, int count)`
	`Task WriteAsync(byte[] buffer, int offset, int count)`
	`void WriteByte(byte value)`
	`void CopyTo(Stream destination)`
	`Task CopyToAsync(Stream destination)`
Перемещение	`bool CanSeek { get; }`
	`long Position { get; set; }`
	`void SetLength(long value)`
	`long Length { get; }`
	`long Seek(long offset, SeekOrigin origin)`
Закрытие потока	`void Close()`
	`void Dispose()`
	`void Flush()`
	`Task FlushAsync()`
Таймауты	`bool CanTimeout { get; }`
	`int ReadTimeout { get; set; }`
	`int WriteTimeout { get; set; }`
Другие члены	`static readonly Stream Null`
	`static Stream Synchronized(Stream stream)`

Класс `Stream` вводит поддержку асинхронных операций ввода/вывода. Для этого служат методы вида *`ИмяОперации`*`Async()` (например, `ReadAsync()`). Такие методы имеют перегруженные версии с дополнительным параметром – токеном отмены[1]. Асинхронный ввод/вывод поддерживается также при помощи методов `BeginRead()` и `BeginWrite()`. Однако, это устаревший шаблон использования асинхронных операций[2].

[1] Подробно о принципах использования асинхронных методов буден рассказано далее.

[2] Данный шаблон частично разбирается в параграфе «Асинхронный вызов методов».

Статический метод `Synchronized()` возвращает оболочку для потока, которая обеспечивает безопасность при совместной работе нескольких *потоков выполнения* (threads).

Использование методов и свойств класса `Stream` будет показано на примере работы с классом `FileStream`. Объект класса `FileStream` возвращается некоторыми методами классов `FileInfo` и `File`. Кроме этого, данный объект можно создать при помощи конструктора с параметрами, включающими имя файла и опции доступа к файлу.

```
// создаём файл test.dat в текущем каталоге и записываем 100 байтов
var fs = new FileStream("test.dat", FileMode.OpenOrCreate);
for (byte i = 0; i < 100; i++)
{
    fs.WriteByte(i);
}

// можно записать информацию из массива байтов
// первый аргумент метода Write() – массив, второй – смещение
// в массиве, третий – количество записываемых байтов
byte[] info = {1, 2, 3, 4, 5, 6, 7, 8, 9};
fs.Write(info, 2, 4);

// возвращаемся на начало потока и читаем все байты
fs.Position = 0;
while (fs.Position < fs.Length)
{
    Console.Write(fs.ReadByte());
}

// закрываем поток (и файл), освобождая ресурсы
fs.Close();
```

Класс `MemoryStream` даёт возможность организовать поток в оперативной памяти. Свойство `Capacity` этого класса позволяет получить или установить количество байтов, выделенных под поток. Метод `ToArray()` записывает все содержимое потока в массив байтов. Метод `WriteTo()` переносит содержимое потока из памяти в другой поток, производный от класса `Stream`.

Класс `BufferedStream` – это декоратор потока для повышения производительности путём буферизации данных. В примере кода `BufferedStream` работает с `FileStream`, предоставляя 20000 байтов буфера. То есть, второе физическое обращение к файлу произойдёт только при чтении 20001-го байта[1].

```
// записываем 100000 байтов в файл
File.WriteAllBytes("myFile.bin", new byte[100000]);
```

[1] Заметим, что класс `FileStream` уже обладает некоторой поддержкой буферизации.

```
// читаем, используя буфер
using (FileStream fs = File.OpenRead("myFile.bin"))
{
    using (var bs = new BufferedStream(fs, 20000))
    {
        bs.ReadByte();
        Console.WriteLine(fs.Position);    // 20000
    }
}
```

Классы `DeflateStream` и `GZipStream` являются декораторами потока, реализующими сжатие данных. Они различаются тем, что `GZipStream` записывает дополнительные данные о протоколе сжатия в начало и конец потока. В следующем примере сжимается и восстанавливается текстовый поток из 1000 слов.

```
// формируем набор из 1000 слов
var words = "The quick brown fox jumps over the lazy dog".Split();
var rnd = new Random();
var text = Enumerable.Repeat(0, 1000)
                     .Select(i => words[rnd.Next(words.Length)]);

// using обеспечит корректное закрытие потоков
using (Stream s = File.Create("compressed.bin"))
{
    using (var ds = new DeflateStream(s, CompressionMode.Compress))
    {
        using (TextWriter w = new StreamWriter(ds))
        {
            foreach (string word in text)
            {
                w.Write(word + " ");
            }
        }
    }
}

using (Stream s = File.OpenRead("compressed.bin"))
{
    using (var ds = new DeflateStream(s,CompressionMode.Decompress))
    {
        using (TextReader r = new StreamReader(ds))
        {
            Console.Write(r.ReadToEnd());
        }
    }
}
```

17.2. Адаптеры потоков

Перейдём к рассмотрению классов-адаптеров для потоков. Классы `BinaryReader` и `BinaryWriter` позволяют при помощи своих методов читать и записывать в поток данные примитивных типов и массивов байтов или символов. Вся информация записывается в поток в двоичном представлении. Рассмотрим работу с этими классами на примере типа `Student`, который может записать свои данные в поток.

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double GPA { get; set; }    // Grade Point Average

    public void SaveBinaryToStream(Stream stream)
    {
        // конструктор позволяет "обернуть" адаптер вокруг потока
        var bw = new BinaryWriter(stream);

        // BinaryWriter имеет 18 перегруженных версий метода Write()
        bw.Write(Name);
        bw.Write(Age);
        bw.Write(GPA);

        // убеждаемся, что буфер BinaryWriter пуст
        bw.Flush();
    }

    public void ReadBinaryFromStream(Stream stream)
    {
        var br = new BinaryReader(stream);
        // для чтения каждого примитивного типа есть свой метод
        Name = br.ReadString();
        Age = br.ReadInt32();
        GPA = br.ReadDouble();
    }
}
```

Абстрактные классы `TextReader` и `TextWriter` дают возможность читать и записывать данные в поток в строковом представлении. При этом имеется поддержка асинхронного выполнения (методы вида *ИмяОперации*`Async()`). От этих классов наследуются классы `StreamReader` и `StreamWriter`. Представим методы для работы с данными класса `Student` с использованием `StreamReader` и `StreamWriter`:

```
public void SaveToStream(Stream stream)
{
    var sw = new StreamWriter(stream);
```

```
        // запись напоминает вывод на консоль (и не случайно)
        sw.WriteLine(Name);
        sw.WriteLine(Age);
        sw.WriteLine(GPA);
        sw.Flush();
    }

    public void ReadFromStream(Stream stream)
    {
        var sr = new StreamReader(stream);

        // читаем данные как строки
        Name = sr.ReadLine();
        Age = Int32.Parse(sr.ReadLine());
        GPA = Double.Parse(sr.ReadLine());
    }
```

18. Основы XML и JSON

Расширяемый язык разметки (eXtensible Markup Language, XML) – это способ описания структурированных данных. *Структурированными данными* называются такие данные, которые обладают заданным набором семантических атрибутов и допускают иерархическое описание. XML-данные содержатся в *документе*, в роли которого может выступать файл, поток или другое хранилище информации, способное поддерживать текстовый формат.

Любой XML-документ строится по определённым правилам. Ниже перечислены правила, следовать которым обязательно.

1. Единица информации – XML-элемент. XML-документ состоит из XML-элементов. Каждый элемент определяется при помощи *имени*, *открывающего тега* и *закрывающего тега*. Открывающий тег элемента записывается в форме `<имя-элемента>`, закрывающий тег – в форме `</имя-элемента>`. Между открывающим и закрывающим тегами размещается *содержимое элемента*. Если содержимое элемента отсутствует, то элемент может быть записан в форме `<имя-элемента />` или `<имя-элемента/>`.

2. Иерархия элементов. Содержимым XML-элемента может быть текст, пробельные символы (пробелы, табуляции, переводы строки), а также другие XML-элементы. Допускается комбинация указанного содержимого (например, элемент может содержать и текст, и вложенные элементы). Элементы должны быть правильно вложены друг в друга – если элемент `A` вложен в элемент `B`, то закрывающий тег `</A>` должен находиться перед закрывающим тегом `</B>`.

3. Корневой элемент. В XML-документе всегда должен быть единственный элемент, называемый *корневым*, никакая часть которого не входит в содержимое любого другого элемента. Иначе говоря, корневой элемент обрамляет все остальные элементы документа.

4. Синтаксис имён элемента. Имена элементов чувствительны к регистру. Имена могут содержать буквы, цифры, дефисы (-), символы подчёркивания (_),

двоеточия (`:`) и точки (`.`), однако должны начинаться только с буквы или символа подчёркивания. Двоеточие может быть использовано только в специальных случаях – при записи префикса пространства имён. Имена, начинающиеся с `xml` (вне зависимости от регистра), зарезервированы для нужд XML.

5. Специальные символы. Некоторые символы не могут использоваться в тексте содержимого элементов, так как применяются в разметке документа. Эти символы могут быть обозначены особым образом:

`&` символ &
`<` символ <
`>` символ >
`&quot;` символ "
`&apos;` символ '
`&#int;` Unicode-символ с десятичным кодом *int*
`&#xhex;` Unicode-символ с шестнадцатеричным кодом *hex*

6. Атрибуты элемента. Любой XML-элемент может содержать один или несколько *атрибутов*, записываемых в открывающем теге. Правила на имена атрибутов накладываются такие же, как и на имена элементов. Имена атрибутов отделяются от их значений символом `=`. Значение атрибута заключается в апострофы или в двойные кавычки. Если апостроф или двойные кавычки присутствуют в значении атрибута, то для обрамления используются те из них, которые не встречаются в значении. Приведём пример элементов с атрибутами:

```
<elements-with-attributes>
  <one attr="value"/>
  <several first="1" second="2" third="3"/>
  <apos-quote case1="John's" case2='He said:"Hello, world!"'/>
</elements-with-attributes>
```

7. Особые части XML-документа. Кроме элементов, XML-документ может содержать *XML-декларацию*, *комментарии*, *инструкции по обработке*, *секции CDATA*.

XML-документ может начинаться с XML-декларации, определяющей используемую версию XML, кодировку XML-документа и наличие внешних зависимостей (обязательным атрибутом является только версия):

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
```

Комментарии размещаются в любом месте документа и записываются в обрамлении `<!--` и `-->`. В тексте комментариев не должна содержаться последовательность из двух знаков дефиса.

XML-документ может содержать инструкции по обработке, несущие информацию для внешних приложений. Инструкции по обработке записываются в обрамлении `<?` и `?>`.

Секция CDATA используется для того, чтобы обозначить части документа, которые не должны восприниматься как разметка. Секция CDATA начинается со

строки `<![CDATA[` и заканчивается строкой `]]>`. Внутри самой секции не должна присутствовать строка `]]>`:

```
<example>
  <![CDATA[ <aaa>bb&cc<<<]]>
</example>
```

Если XML-документ оформлен по описанным выше правилам, то он называется *правильно построенным документом* (well-formed document). Если правильно построенный XML-документ удовлетворяет некой семантической схеме, задающей его структуру и содержание, то он называется *действительным документом* (valid document).

Приведём XML-документ с описанием планет, который будет использован далее в примерах кода:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- первые четыре планеты -->
<planets>
  <planet>
    <name>Mercury</name>
  </planet>
  <planet>
    <name>Venus</name>
  </planet>
  <planet>
    <name>Earth</name>
    <moon>
      <name>Moon</name>
      <period units="days">27.321582</period>
    </moon>
  </planet>
  <planet>
    <name>Mars</name>
    <moon>
      <name>Phobos</name>
      <period units="days">0.318</period>
    </moon>
    <moon>
      <name>Deimos</name>
      <period units="days">1.26244</period>
    </moon>
  </planet>
</planets>
```

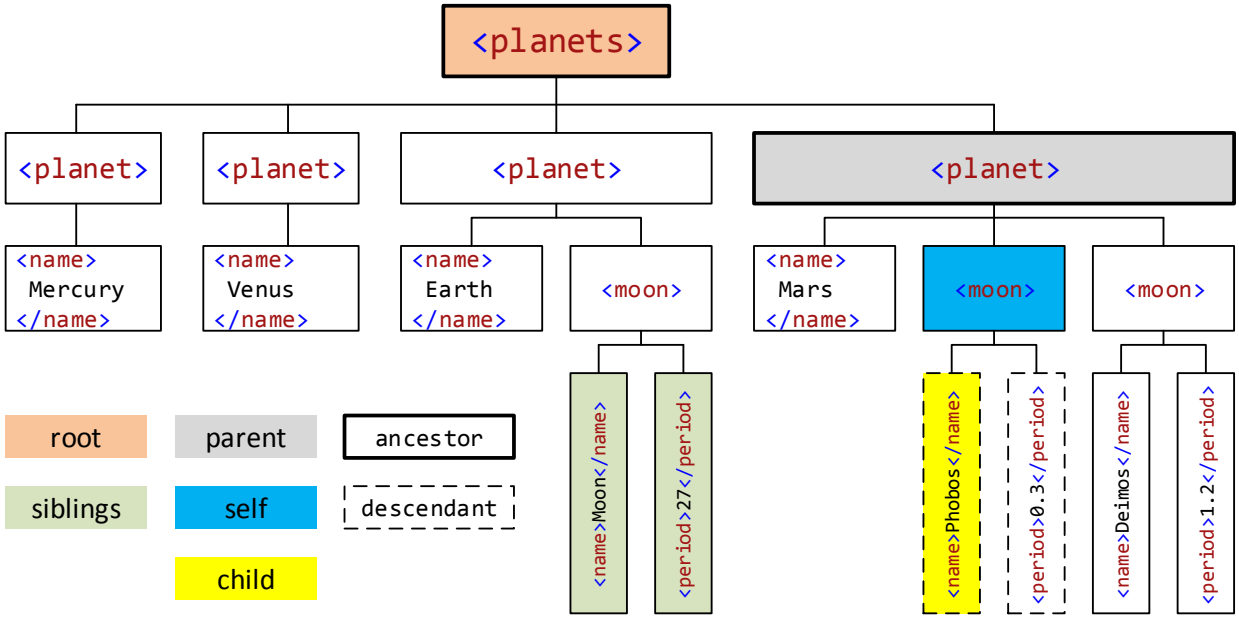

Рис. 5. Древовидная структура XML-данных.

Отметим, что при описании дерева XML-элементов используются следующие термины (см. рис. 5):

1. *Текущий элемент* (self);
2. *Предок* (ancestor) – любой элемент, содержащий текущий;
3. *Корень* (root) – предок всех элементов;
4. *Родитель* (parent) – непосредственный предок текущего элемента;
5. *Потомок* (descendant) – любой элемент, вложенный в текущий;
6. *Ребёнок* (child) – непосредственный потомок текущего элемента;
7. *Сиблинги* (siblings) – элементы, имеющие общего родителя.

В XML-документе с описанием планет элемент `<name>` используется и как имя планеты, и как имя луны. Ссылки на `<name>` будут неоднозначны – два одинаковых имени несут разную смысловую нагрузку. Для устранения неоднозначности и обеспечения семантической уникальность элемента предназначены *пространства имён XML*.

Чтобы связать элемент с пространством имён, используется специальный атрибут `xmlns`. Обычно для идентификатора пространства имён используют *унифицированный идентификатор ресурса* (Uniform Resource Identifier, URI), чтобы уменьшить риск совпадения идентификаторов в разных документах.

```
<!-- фрагмент документа с планетами -->
<planet>
  <name xmlns="http://astronomy.com/planet" >Earth</name>
  <moon>
    <name xmlns="http://astronomy.com/moon" >Moon</name>
    <period units="days">27.321582</period>
  </moon>
  </planet>
```

Чтобы не задавать атрибут `xmlns` у каждого элемента, действуют следующие правила. Считается, что пространство имён, заданное у элемента, автоматически распространяется на все дочерние элементы. Также при описании пространства имён можно определить *префикс*, который затем записывается перед именем требуемых элементов через двоеточие. Атрибуты также могут быть связаны с пространствами имён при помощи префиксов.

```
<!-- пространства имён обычно определяют в начале документа -->
<planets xmlns="http://astronomy.com/planet"
         xmlns:m="http://astronomy.com/moon">
  <planet>
    <name>Earth</name>
    <m:moon>
      <m:name>Moon</m:name>
      <m:period m:units="days">27.321582</m:period>
    </m:moon>
  </planet>
</planets>
```

JSON (JavaScript Object Notation) – простой текстовый формат обмена данными, удобный для чтения и записи как человеком, так и компьютером. Он основан на подмножестве языка программирования JavaScript.

JSON строится на двух структурах:

– *Коллекция пар ключ/значение*. В разных языках, эта концепция реализована как объект, запись, структура, словарь или ассоциативный массив.

– *Упорядоченный список значений*. В большинстве языков это реализовано как массив, вектор, список или последовательность.

В нотации JSON эти структуры принимают следующие формы:

1. *Объект* – неупорядоченный набор пар ключ/значение. Объект начинается с открывающей фигурной скобки и заканчивается закрывающей фигурной скобкой. Каждое имя ключа сопровождается двоеточием, пары ключ/значение разделяются запятой.

2. *Массив* – упорядоченная коллекция значений. Массив начинается с открывающей квадратной скобки и заканчивается закрывающей квадратной скобкой. Значения разделены запятой.

Значение может быть *строкой*, *числом*, `true`, `false`, `null`, *объектом* или *массивом*. Эти структуры могут быть вложенными. *Строка* – коллекция нуля или больше символов Unicode, заключённая в двойные кавычки. Число представляется так же, как в C# (используется только десятичная система счисления). Пробелы могут использоваться между любыми лексемами. Исключая некоторые детали кодирования, вышеизложенное полностью описывает JSON[1].

Приведём пример простейшего JSON-документа (описание объекта, представляющего информацию о студенте с набором оценок):

[1] Подробности и примеры можно найти на сайте json.org.

```
{
  "id": 12,
  "name": "Petrov",
  "marks": [10, 8, 7, 9]
}
```

19. Технология LINQ to XML

Технология LINQ to XML предоставляет программный интерфейс для работы с XML-документами, описываемыми в виде дерева объектов. Этот программный интерфейс обеспечивает создание, редактирование и трансформацию XML, при этом возможно применение LINQ-подобного синтаксиса.

LINQ to XML содержит набор классов, сосредоточенных в пространстве имён `System.Xml.Linq` (рис. 6):

1. Абстрактные классы `XObject`, `XNode` и `XContainer` служат основой для иерархии классов, соответствующих различным объектам XML.
2. Классы `XElement` и `XDocument` представляют XML-элемент и XML-документ соответственно.
3. Класс `XAttribute` служит для описания XML-атрибута.
4. Класс `XText` представляет текст в XML-элементе.
5. Класс `XName` представляет имя атрибута или элемента.
6. Классы `XDeclaration`, `XDocumentType`, `XProcessingInstruction`, `XComment` описывают соответствующие части XML-документа.
7. Статический класс `Extensions` содержит методы расширения для выполнения запросов к XML-данным.

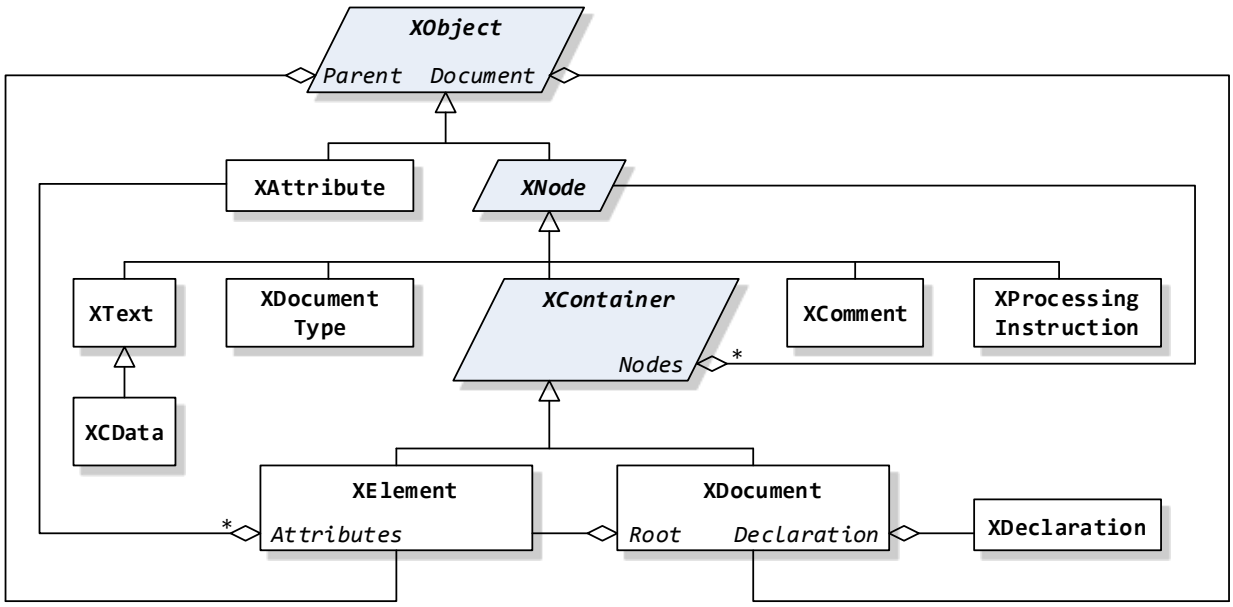

Рис. 6. Основные классы LINQ to XML.

Центральную роль в работе с XML-данными играют классы `XElement`, `XDocument` и `XAttribute`. Для создания отдельного XML-элемента обычно используется один из конструкторов класса `XElement`:

```
public XElement(XElement other);
public XElement(XName name);
public XElement(XStreamingElement other);
public XElement(XName name, object content);
public XElement(XName name, params object[] content);
```

Первая версия конструктора `XElement` является копирующим конструктором, а вторая – создаёт пустой элемент. Обсудим использование четвёртой версии, которая позволяет указать имя элемента и его содержимое. Заметим, что существует неявное преобразование строки в `XName`. Интерпретация аргумента `content` производится по табл. 14.

Таблица 14

Интерпретация аргумента `content` конструктора класса `XElement`

Тип или значение content	**Способ обработки**
`string`	Преобразуется в дочерний объект типа `XText` и добавляется как текстовое содержимое элемента
`XText`	Добавляется как дочерний объект – текстовое содержимое элемента
`XElement`	Добавляется как дочерний элемент
`XAttribute`	Добавляется как атрибут элемента
`XProcessingInstruction`, `XComment`	Добавляется как дочернее содержимое. При добавлении объектов типа `XAttribute` или унаследованных от `XNode` выполняется клонирование, если эти объекты уже имеют родителя в дереве объектов XML.
`IEnumerable`	Объект перечисляется и обрабатывается рекурсивно. Коллекция строк добавляется в виде единого текста
`Null`	Этот объект игнорируется
Любой прочий тип	Вызывается метод `ToString()`, и результат трактуется как `string`

Ниже приведены различные варианты вызова конструктора `XElement`:

```
var e1 = new XElement("name", "Earth");
// <name>Earth</name>

var e2 = new XElement("planet", e1);
// <planet>
//   <name>Earth</name>
// </planet>

var e3 = new XElement("period", new XAttribute("units", "days"));
// <period units="days" />

var e4 = new XElement("comment", new XComment("the comment"));
// <comment>
//   <!--the comment-->
// </comment>

var e5 = new XElement("list", new List<object> {"text",
                                    new XElement("name", "Mars")});
```

```
// <list>
//   text<name>Mars</name>
// </list>

var e6 = new XElement("moon", null);
// <moon />

var e7 = new XElement("date", DateTime.Now);
// <date>2010-01-19T11:04:54.625+02:00</date>
```

Пятая версия конструктора `XElement` подобна четвёртой, но позволяет предоставить множество объектов для содержимого:

```
var p = new XElement("planets",
            new XElement("planet", new XElement("name", "Mercury")),
            new XElement("planet", new XElement("name", "Venus")),
            new XElement("planet", new XElement("name", "Earth"),
                new XElement("moon",
                    new XElement("name", "Moon"),
                    new XElement("period", 27.321582,
                                 new XAttribute("units", "days")))));
Console.WriteLine(p);    // выводим первые три планеты в виде XML
```

Использование конструктора `XAttribute` для создания XML-атрибута очевидно из приведённых выше примеров. Конструкторы класса `XDocument` позволяют указать декларацию XML-документа и набор объектов содержимого. В этот набор могут входить комментарии, инструкции по обработке и не более одного XML-элемента:

```
var d = new XDeclaration("1.0", "utf-8", "yes");
// используем элемент p из предыдущего примера
var doc = new XDocument(d, new XComment("первые три планеты"), p);
Console.WriteLine(doc.Declaration);    // печатаем декларацию
Console.WriteLine(doc);                // печатаем документ
```

Кроме применения конструкторов, объекты XML можно создать из строкового представления при помощи статических методов `XElement.Parse()` и `XDocument.Parse()`:

```
var planet = XElement.Parse("<name>Earth</name>");
```

Для сохранения элемента или XML-документа используется метод `Save()`, имеющийся у `XElement` и `XDocument`. Данный метод перегружен и позволяет выполнить запись в текстовый файл или с применением адаптеров `TextWriter` и `XmlWriter`. Кроме этого, можно указать опции сохранения (например, отключить автоматическое формирование отступов элементов).

```
doc.Save("planets.xml", SaveOptions.None);
```

Загрузка элемента или XML-документа выполняется статическими методами `XElement.Load()` или `XDocument.Load()`. Метод `Load()` перегружен и позволяет выполнить загрузку из файла, произвольного URI, а также с применением адаптеров `TextReader` и `XmlReader`. Можно задать опции загрузки (например, связать с элементами XML номер строки в исходном тексте).

```
var d1 = XDocument.Load("planets.xml", LoadOptions.SetLineInfo);
var d2 = XElement.Load("http://habrahabr.ru/rss/");
```

Рассмотрим набор методов и свойств, используемых в LINQ to XML при выборке данных. Класс `XObject` имеет свойство `NodeType` для типа XML-узла и свойство `Parent`, указывающее, какому элементу принадлежит узел.

Методы классов `XNode` и `XContainer` позволяют получить у элемента наборы предков и дочерних узлов (элементов). При этом возможно указание фильтра – имени элемента. Большинство методов возвращают коллекции, реализующие `IEnumerable`[1].

```
// методы выборки у XNode
public IEnumerable<XElement> Ancestors();            // + XName name
public IEnumerable<XElement> ElementsAfterSelf();    // + XName name
public IEnumerable<XElement> ElementsBeforeSelf();   // + XName name
public bool IsAfter(XNode node);
public bool IsBefore(XNode node);
public IEnumerable<XNode> NodesAfterSelf();
public IEnumerable<XNode> NodesBeforeSelf();

// методы выборки у XContainer
public IEnumerable<XNode> DescendantNodes();
public IEnumerable<XElement> Descendants();          // + XName name
public XElement Element(XName name);
public IEnumerable<XElement> Elements();             // + XName name
```

Класс `XDocument` позволяет получить корневой элемент при помощи свойства `Root`. В классе `XElement` есть методы для запроса элементов-предков и элементов-потомков, а также методы для запроса атрибутов.

```
// методы выборки у XElement
public IEnumerable<XElement> AncestorsAndSelf();     // + XName name
public XAttribute Attribute(XName name);
public IEnumerable<XAttribute> Attributes();         // + XName name
public IEnumerable<XNode> DescendantNodesAndSelf();
public IEnumerable<XElement> DescendantsAndSelf();   // + XName name
```

Статический класс `Extensions` определяет несколько методов расширения, работающих с коллекциями элементов или узлов XML:

---

[1] Запись `+ XName name` означает наличие перегруженной версии метода с параметром `name` типа `XName`.

```
IEnumerable<XElement> Ancestors<T>(...) where T: XNode;
IEnumerable<XElement> AncestorsAndSelf(...);
IEnumerable<XAttribute> Attributes(...);
IEnumerable<XNode> DescendantNodes<T>(...) where T: XContainer;
IEnumerable<XNode> DescendantNodesAndSelf(...);
IEnumerable<XElement> Descendants<T>(...) where T: XContainer;
IEnumerable<XElement> DescendantsAndSelf(...);
IEnumerable<XElement> Elements<T>(...) where T: XContainer;
IEnumerable<T> InDocumentOrder<T>(...) where T: XNode;
IEnumerable<XNode> Nodes<T>(...) where T: XContainer;
void Remove(this IEnumerable<XAttribute> source);
void Remove<T>(this IEnumerable<T> source) where T: XNode;
```

Продемонстрируем примеры запросов, используя файл `planets.xml` с описанием четырёх планет. Загрузим файл и выведем имена планет:

```
var xml = XDocument.Load("planets.xml");
var query = xml.Root.Elements("planet")
                    .Select(planet => planet.Element("name").Value);
foreach (string name in query)
    Console.WriteLine(name);
```

Выберем все элементы с именем `"moon"`, которые являются потомками корневого элемента. У каждого из элементов возьмём ребёнка с именем `"name"`:

```
var moons = xml.Descendants("moon").Select(p => p.Element("name"));
```

Найдём элемент, содержащий указанный текст:

```
var phobos = xml.DescendantNodes().OfType<XText>()
                .Where(text => text.Value == "Phobos")
                .Select(text => text.Parent);
```

Модификация XML-информации в памяти выполняется при помощи следующих методов классов `XNode`, `XContainer`, `XElement`[1]:

```
// методы модификации у XNode
public void AddAfterSelf(object content);                 // + params
public void AddBeforeSelf(object content);                // + params
public void Remove();
public void ReplaceWith(object content);                  // + params

// методы модификации у XContainer
public void Add(object content);                          // + params
public void AddFirst(object content);                     // + params
public void RemoveNodes();
public void ReplaceNodes(object content);                 // + params
```

[1] Запись `+ params` означает наличие перегруженной версии метода с параметром типа `params object[]`.

```
// методы модификации у XElement
public void RemoveAll();
public void RemoveAttributes();
public void ReplaceAll(object content);                    // + params
public void ReplaceAttributes(object content);             // + params
public void SetAttributeValue(XName name, object value);
public void SetElementValue(XName name, object value);
```

Рассмотрим несколько примеров модификации XML. Начнём с удаления заданных узлов. Для этого используем метод расширения `Remove<T>()`:

```
var xml = XDocument.Load("planets.xml");
xml.Element("planets").Elements("planet").Elements("moon").Remove();
```

Продемонстрируем добавление и замену элементов:

```
var p = XDocument.Load("planets.xml").Root;
p.Add(new XElement("planet", new XElement("name", "Jupiter")));
var moon = p.Descendants("moon").First();
moon.ReplaceWith(new XElement("DeathStar"));
```

Покажем добавление атрибута к элементу:

```
XElement sun = xml.Element("planets");
sun.Add(new XAttribute("MassOfTheSun", "332.946 Earths"));
```

Функциональные конструкторы и поддержка методами выборки коллекций открывают богатые возможности трансформации наборов объектов в XML. В следующем примере показано создание XML-документа на основе коллекции студентов `gr` (коллекция описана в параграфе, посвящённом LINQ to objects):

```
var xml = new XElement("students",
          from student in gr
          select new XElement("student",
                     new XAttribute("name", student.Name),
                     new XAttribute("age", student.Age),
                     from mark in student.Marks
                     select new XElement("mark", mark)));
```

Хотя предыдущий пример использует отложенный оператор `Select()`, полная итерация по набору выполняется конструктором класса `XElement`. Если необходимо отложенное конструирование XML-элементов, следует воспользоваться классом `XStreamingElement`.

```
string[] names = {"John", "Paul", "George", "Pete"};
var xml = new XStreamingElement("Beatles",
                            from n in names
                            select new XStreamingElement("name", n));
names[3] = "Ringo";         // это присваивание изменит объект xml
```

Для описания пространства имён XML в LINQ to XML используется класс `XNamespace`. У этого класса нет открытого конструктора, но определено неявное приведение строки к `XNamespace`:

```
XNamespace ns = "http://astronomy.com/planet";
```

Чтобы указать на принадлежность имени к определённому пространству имён, следует использовать перегруженную версию оператора +, объединяющую объект `XNamespace` и строку в результирующий объект `XName`:

```
XElement jupiter = new XElement(ns + "name", "Jupiter");
// <name xmlns="http://astronomy.com/planet">Jupiter</name>
```

Префикс пространства имён устанавливается путём добавления в элемент атрибута специального вида. Если префикс задан, им заменяется любое указание пространства имён у дочернего элемента:

```
XElement planet = new XElement(ns + "planet",
                      new XAttribute(XNamespace.Xmlns + "p", ns));
planet.Add(new XElement(ns + "name", "Jupiter"));
Console.WriteLine(planet);
// <p:planet xmlns:p="http://astronomy.com/planet">
//   <p:name>Jupiter</p:name>
// </p:planet>
```

20. Дополнительные возможности обработки XML

В дополнение к LINQ to XML, платформа .NET содержит несколько программных интерфейсов для работы с XML. Для этого обычно используются классы из пространств имён вида `System.Xml.*` (сборка `System.Xml.dll`).

Классы `XmlReader` и `XmlWriter` – это основа механизма последовательного чтения, обработки и записи XML-документов. Такой подход выгодно использовать, когда документ слишком велик, чтобы читать его в память целиком, или содержит ошибки в структуре.

Для чтения XML-документов применяется класс `XmlReader` и его наследники `XmlTextReader` (чтение на основе текстового потока), `XmlNodeReader` (разбор XML из объектов `XmlNode`) и `XmlValidatingReader` (чтение с проверкой схемы XML-документа). Класс `XmlReader` содержит статический метод `Create()`, создающий объект для чтения на основе `Stream`, `TextReader` или строки URI (частный случай URI – имя файла):

```
XmlReader reader = XmlReader.Create("planets.xml");
```

Метод `Create()` принимает в качестве дополнительного аргумента объект класса `XmlReaderSettings`, который задаёт различные опции чтения данных:

```
// игнорируем при чтении комментарии и пробельные символы
var settings = new XmlReaderSettings();
```

```
settings.IgnoreComments = true;
settings.IgnoreWhitespace = true;
XmlReader reader = XmlReader.Create("planets.xml", settings);
```

Объект `XmlReader` извлекает XML-конструкции из документа при помощи метода `Read()`[1]. Тип текущей конструкции можно узнать, используя свойство `NodeType`, значениями которого являются элементы перечисления `XmlNodeType`. В табл. 15 приведены основные элементы этого перечисления. С конструкцией работают, применяя свойства `Name` (имя элемента), `Value` (данные элемента) и некоторые другие.

Таблица 15

Основные элементы перечисления `XmlNodeType`

Значение	Пример
CDATA	`<![CDATA[ This is CDATA info ]]>`
Comment	`<!-- первые четыре планеты -->`
Document	`<planets>` (корневой элемент)
Element	`<planet>`
EndElement	`</planet>`
ProcessingInstruction	`<?perl lower-to-upper-case ?>`
Text	Mercury
Whitespace	\r \t \n (перевод строки, табуляция)
XmlDeclaration	`<?xml version="1.0" encoding="utf-8" ?>`

Следующий пример демонстрирует разбор XML-файла и печать разобранных конструкций:

```
using (var r = XmlReader.Create("planets.xml"))
{
    while (r.Read())
    {
        Console.Write("{0}\t", r.Depth);
        switch (r.NodeType)
        {
            case XmlNodeType.Element:
            case XmlNodeType.EndElement:
                Console.WriteLine(r.Name);
                break;
            case XmlNodeType.Text:
            case XmlNodeType.Comment:
            case XmlNodeType.XmlDeclaration:
                Console.WriteLine(r.Value);
                break;
        }
    }
}
```

[1] У класса `XmlReader` имеются также специфичные методы чтения конкретного содержимого XML-документа (например, `ReadContentAsInt()`, `ReadAttributeValue()`).

Для чтения атрибутов текущего элемента можно использовать индексатор XmlReader, указав имя или позицию атрибута (если атрибута не существует, индексатор вернёт значение null):

```
// переписанный фрагмент оператора switch из предыдущего примера
switch (r.NodeType)
{
    case XmlNodeType.Element:
        Console.WriteLine(r.Name);
        string attribute = r["units"];
        Console.WriteLine(attribute ?? string.Empty);
        break;
    . . .
}
```

Набор методов класса XmlReader, начинающихся с префикса MoveTo (MoveToElement() и т. п.), может использоваться для перехода к следующей XML-конструкции в потоке. Вернуться к просмотренным конструкциям нельзя.

Класс XmlWriter – это абстрактный класс для создания XML-данных. Подчеркнём, что XML-данные всегда могут быть сформированы в виде простой строки и затем записаны в любой поток. Однако такой подход не лишён недостатков – возрастает вероятность неправильного формирования структуры XML из-за элементарных ошибок. Класс XmlWriter и его наследники (например, XmlTextWriter) предоставляют более «помехоустойчивый» способ генерации XML-документа.

Приведём пример работы с классом XmlWriter.

```
// опция для формирования отступов в документе
var settings = new XmlWriterSettings {Indent = true };
using (var writer = XmlWriter.Create("customers.xml", settings))
{
    // начинаем с XML-декларации
    writer.WriteStartDocument();

    // открывающий тег с двумя атрибутами
    writer.WriteStartElement("customer");
    writer.WriteAttributeString("id", "1");
    writer.WriteAttributeString("status", "archived");

    // вложенный элемент со строковым содержимым
    writer.WriteElementString("name", "Alex");

    // так пишутся элементы с не строковым содержимым
    // для этого используется метод WriteValue()
    writer.WriteStartElement("birthdate");
    writer.WriteValue(new DateTime(1975, 8, 4));
    writer.WriteEndElement();
```

```
    // закрывающие теги (принцип стека)
    writer.WriteEndElement();
    writer.WriteEndDocument();
}
```

Этот пример формирует следующий XML-документ:

```
<?xml version="1.0" encoding="utf-8"?>
<customer id="1" status="archived">
  <name>Alex</name>
  <birthdate>1975-08-04T00:00:00</birthdate>
</customer>
```

Классы `XmlNode`, `XmlAttribute`, `XmlElement`, `XmlDocument` служат для представления XML-документа в виде дерева объектов. Программный интерфейс, основанный на использовании данных классов, являлся предшественником LINQ to XML. В связи с этим ограничимся только простым примером, демонстрирующим работу с указанными классами:

```
public static void OutputNode(XmlNode node)
{
    Console.WriteLine("Type= {0} \t Name= {1} \t Value= {2}",
                      node.NodeType, node.Name, node.Value);
    if (node.Attributes != null)
    {
        foreach (XmlAttribute attr in node.Attributes)
        {
            Console.WriteLine("Type={0} \t Name={1} \t Value={2}",
                              attr.NodeType, attr.Name, attr.Value);
        }
    }
    // если есть дочерние элементы, рекурсивно обрабатываем их
    if (node.HasChildNodes)
    {
        foreach (XmlNode child in node.ChildNodes)
        {
            OutputNode(child);
        }
    }
}

// пример использования метода OutputNode()
var doc = new XmlDocument();
doc.Load("planets.xml");
OutputNode(doc.DocumentElement);
```

21. Сериализация времени выполнения

Сериализация времени выполнения (runtime serialization) – это процесс преобразования объекта или графа связанных объектов в поток байтов. *Десериализация* – обратный процесс, заключающийся в восстановлении состояния объекта из потока байтов. Возможности сериализации используются при сохранении состояния объекта в файле, передаче объектов по сети, клонировании объектов.

Платформа .NET обладает встроенным механизмом поддержки сериализации времени выполнения. Основные классы, связанные с сериализацией, размещены в пространствах имён с префиксом `System.Runtime.Serialization`. Например, сериализацию в двоичном формате обеспечивают классы из `System.Runtime.Serialization.Formatters.Binary`.

При рассмотрении примеров сериализации будем использовать классы, описывающие одного студента и группу студентов:

```
public class Student : IComparable<Student>
{
    private string _name;
    private double _gpa;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public double GPA    // Grade Point Average, средний балл
    {
        get { return _gpa; }
        set { _gpa = value; }
    }

    public int CompareTo(Student other)
    {
        return GPA.CompareTo(other.GPA);
    }
}

public class Group : Collection<Student>
{
    private Student _bestStudent;
    private double _gpa;

    public double GPA
    {
        get { return _gpa; }
    }

    public Student BestStudent
```

```
        {
            get { return _bestStudent; }
        }

        public double CalculateGroupGPA()
        {
            return _gpa = Items.Select(stud => stud.GPA).Average();
        }

        public Student FindTheBestStudent()
        {
            return _bestStudent = Items.Max();
        }
    }

    // создадим объекты классов Student и Group
    var group = new Group {
                    new Student {Name = "Smirnov", GPA = 9.1},
                    new Student {Name = "Ivanova", GPA = 6.7},
                    new Student {Name = "Sokolov", GPA = 7.6},
                    new Student {Name = "Lebedeva", GPA = 9}};
```

Сериализация времени выполнения применима к объектам сериализуемых типов. *Сериализуемый тип* – это тип, помеченный атрибутом `[Serializable]`[1], у которого все поля имеют сериализуемый тип. Базовые типы платформы .NET являются сериализуемыми. Если планируется сериализация объекта `group`, необходимо добавить атрибут `[Serializable]` к классам `Group` и `Student`. Сериализация некоторых полей может не иметь смысла (например, эти поля вычисляются при работе с объектом или хранят конфиденциальные данные). Для таких полей можно применить атрибут `[NonSerialized]`.

Изменим код классов `Group` и `Student` с учётом вышесказанного:

```
    [Serializable]
    public class Student : IComparable<Student>
    {
        // неизменившиеся элементы класса не показаны
    }

    [Serializable]
    public class Group : Collection<Student>
    {
        // считаем, что поле _bestStudent является вычислимым
        [NonSerialized]
        private Student _bestStudent;
        // неизменившиеся элементы класса не показаны
    }
```

[1] Подробно о синтаксисе применения атрибутов и операторе `typeof` рассказывается далее.

Объекты сериализуемых типов можно сохранить в поток в различных форматах. В частности, платформа .NET поддерживает двоичный формат при помощи класса `BinaryFormatter`. Его экземплярный метод `Serialize()` принимает два аргумента – поток сериализации и сериализуемый объект:

```
var formatter = new BinaryFormatter();
using (Stream s = File.Create("group.dat"))
    formatter.Serialize(s, group);
```

Метод `Deserialize()` класса `BinaryFormatter` выполняет десериализацию:

```
var formatter = new BinaryFormatter();
using (Stream s = File.OpenRead("group.dat"))
    group = (Group) formatter.Deserialize(s);
```

Метод `Deserialize()` размещает объект в памяти и возвращает ссылку на него. При этом конструкторы не вызываются. Это может стать проблемой, если нужна особая инициализация объекта и восстановление несохраненных полей. Платформа .NET содержит атрибуты `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]`, `[OnDeserialized]`, применимые к методам сериализуемого типа. Помеченные методы вызываются CLR автоматически до и после сериализации или десериализации соответственно. Метод, который обозначен одним из указанных атрибутов, должен принимать в качестве аргумента объект класса `StreamingContext`[1] и не возвращать значений. Каждый из атрибутов может применяться только к одному методу в типе.

```
[Serializable]
public class Group : Collection<Student>
{
    [OnSerializing]
    private void BeforeSerialization(StreamingContext context)
    {
        CalculateGroupGPA();
    }

    [OnDeserialized]
    private void AfterDeserialization(StreamingContext context)
    {
        FindTheBestStudent();
    }
    // неизменившиеся элементы класса не показаны
}
```

[1] Класс `StreamingContext` описывает контекст потока сериализации. Основным свойством класса является `State`, принимающее значения из перечисления `StreamingContextStates`.

Атрибут [OptionalField] применяется к полю и подавляет генерацию исключения при десериализации, если помеченное поле не найдено в потоке данных. Это позволяет сохранять «старые» объекты, затем модифицировать тип, расширяя состав его полей, и десериализовать данные в «новые» объекты типа.

Если программиста не устраивает способ организации потока сериализуемых данных, он может повлиять на этот процесс, реализуя в сериализуемом типе интерфейс ISerializable:

```
public interface ISerializable
{
    void GetObjectData(SerializationInfo info,
                       StreamingContext context);
}
```

Интерфейс ISerializable позволяет выполнить любые действия, связанные с формированием данных для сохранения. Метод GetObjectData() вызывается CLR автоматически при выполнении сериализации. Реализация метода подразумевает заполнение объекта SerializationInfo набором данных вида «ключ-значение», которые (обычно) соответствуют полям сохраняемого объекта. Класс SerializationInfo содержит перегруженный метод AddValue(), набор методов вида Get*ПримитивныйТип*(), а также свойства для указания имени типа и сборки сериализуемого объекта. Если тип реализует интерфейс ISerializable, он должен содержать специальный private-конструктор, который будет вызывать CLR после выполнения десериализации. Конструктор должен иметь параметры типа SerializationInfo и StreamingContext.

Рассмотрим пример реализации ISerializable в классе Student.

```
[Serializable]
public class Student : IComparable<Student>, ISerializable
{
    void ISerializable.GetObjectData(SerializationInfo info,
                                     StreamingContext ctx)
    {
        info.SetType(typeof (Student));
        info.AddValue("Name", _name);
        info.AddValue("GPA", (int) (_gpa*10));
    }

    private Student(SerializationInfo info, StreamingContext ctx)
    {
        _name = info.GetString("Name");
        _gpa = info.GetInt32("GPA")/10.0;
    }

    public Student() { }

    // неизменившиеся элементы класса не показаны
}
```

22. Сериализация контрактов данных

Контракт данных – это тип (класс или структура), объект которого описывает информационный фрагмент. Подразумевается, что этот фрагмент может быть сохранён, а затем восстановлен. Работу с контрактами данных можно рассматривать как один из механизмов сериализации.

Если в качестве контракта данных используется обычный класс, информационный фрагмент образуют открытые поля и свойства класса. Можно пометить тип атрибутом `[DataContract]`. Тогда информационный фрагмент будут составлять поля и свойства, имеющие атрибут `[DataMember]`[1]. Видимость элементов при этом роли не играет.

```
[DataContract]
public class Student
{
    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public double GPA { get; set; }
}
```

Основным форматом хранения контрактов данных является XML. Поэтому атрибут `[DataContract]` имеет свойства `Name` и `Namespace` для указания имени и пространства имён корневого XML-элемента. У атрибута `[DataMember]` есть свойство `Name`, а также свойства `Order` (порядок записи элементов контракта), `IsRequired` (обязательный элемент для записи), `EmitDefaultValue` (нужна ли запись значения по умолчанию для элемента).

```
[DataContract(Name = "student", Namespace = "bsuir.by")]
public class Student
{
    [DataMember(Name = "name", Order = 1)]
    public string Name { get; set; }

    [DataMember(Name = "gpa", Order = 0)]
    public double GPA { get; set; }
}
```

Если контракт является коллекцией объектов (как класс `Group`), он маркируется атрибутом `[CollectionDataContract]`. Кроме этого, для методов контракта данных применимы атрибуты `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]`, `[OnDeserialized]`.

Если контракт планируется десериализовать в объекты потомков своего типа, эти потомки должны быть упомянуты при помощи атрибута `[KnownType]`.

---

[1] Эти атрибуты размещены в пространстве имён `System.Runtime.Serialization` и одноимённой сборке.

```
[DataContract]
[KnownType(typeof (Postgraduate))]
public class Student { . . . }

public class Postgraduate : Student { . . . }
```

Для выполнения сериализации контракта данных используются классы:
1. `DataContractSerializer` – сериализует контракт в формате XML;
2. `NetDataContractSerializer` – сериализует данные и тип контракта;
3. `DataContractJsonSerializer` – сериализует контракт в формате JSON.

Рассмотрим пример сериализации контрактов данных:

```
var student = new Student {Name = "Smirnov", GPA = 9.1};

// конструктор требует указания типа контракта данных
var ds = new DataContractSerializer(typeof (Student));
using (Stream s = File.Create("smirnov.xml"))
{
    ds.WriteObject(s, student);
}

// полученный файл smirnov.xml
// <student xmlns="bsuir.by"
//          xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
//   <gpa>9.1</gpa>
//   <name>Smirnov</name>
// </student>
```

Сериализацию в формате XML можно выполнить при помощи класса `XmlSerializer` из пространства имён `System.Xml.Serialization`. При таком подходе сохраняются `public`-элементы объекта. Кроме этого, тип объекта должен быть открытым и иметь открытый конструктор без параметров.

```
var student = new Student {Name = "Smirnov", GPA = 9.1};

// при создании XmlSerializer требуется указать сериализуемый тип
var serializer = new XmlSerializer(typeof (Student));

// сериализация
using (Stream stream = File.Create("student.xml"))
{
    serializer.Serialize(stream, student);
}

// десериализация
using (Stream stream = File.OpenRead("student.xml"))
{
    student = (Student) serializer.Deserialize(stream);
}
```

Настройка XML-сериализации может быть выполнена при помощи атрибутов. [XmlRoot] применяется к типу и задаёт корневой элемент в XML-файле. При помощи [XmlElement] настраивается имя и пространство имён XML-элемента. [XmlAttribute] используется, если член класса требуется сохранить в виде XML-атрибута. Поля и свойства, которые не должны сохраняться, помечаются атрибутом [XmlIgnore]. Если тип содержит коллекцию объектов, то настройка имени этой коллекции и имени отдельного элемента выполняется при помощи атрибутов [XmlArray] и [XmlArrayItem].

```
public class Student
{
    [XmlAttribute("name")]
    public string Name { get; set; }

    [XmlIgnore]
    public double GPA { get; set; }
}

[XmlRoot("students")]
public class Group
{
    [XmlArray("list")]
    [XmlArrayItem("student")]
    public List<Student> List { get; set; }
}
```

Отметим, что XML-сериализация не может сохранить коллекции-словари. Если выполняется XML-сериализация объекта, реализующего интерфейс IEnumerable (обычный или универсальный), сохраняются только те элементы, которые доступны через перечислитель.

23. Состав и взаимодействие сборок

В платформе .NET *сборка* (assembly) – это единица развёртывания и контроля версий. Сборка состоит из одного или нескольких *программных модулей* и, возможно, *данных ресурсов*. Программный модуль содержит метаданные и код на Common Intermediate Language. *Метаданные* – это информационные таблицы с полным описанием всех типов, которые размещены в модуле или на которые ссылается модуль. В одном из модулей метаданные хранят *манифест* с описанием всех файлов сборки. Будем называть сборку *однофайловой*, если она состоит из одного файла. Иначе сборка называется *многофайловой*. Тот файл, который содержит манифест сборки, будем называть *главным файлом сборки*. На рис. 7 показана однофайловая сборка `OneFile.exe`, которая ссылается на многофайловую сборку `ManyFiles.dll`.

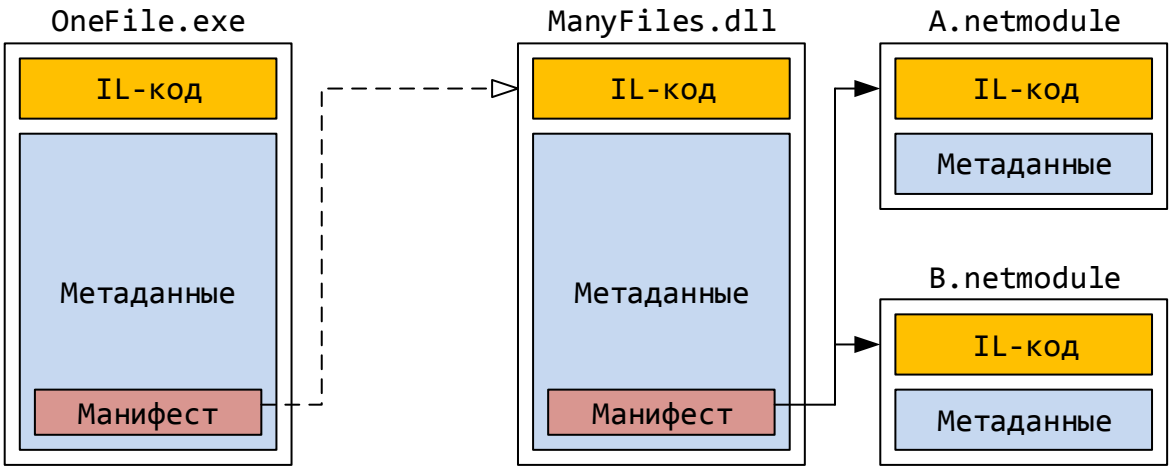

Рис. 7. Однофайловая и многофайловая сборки.

Простые приложения обычно представлены однофайловыми сборками. При разработке сложных приложений переход к многофайловым сборкам даёт следующие преимущества:

– ресурсы (текстовые строки, изображения и т. п.) можно хранить отдельно от исполняемого кода, что позволяет изменять ресурсы без перекомпиляции приложения;

– если исполняемый код приложения разделён на несколько модулей, то модули загружаются в память только по мере надобности;

– скомпилированный модуль может использоваться в нескольких сборках.

Рассмотрим пример создания и использования многофайловой сборки[1]. Пусть требуется построить консольное приложение, в котором метод `Main()` печатает на экране строку. Предположим, что эту строку возвращает статический метод `GetText()` класса `TextClass`.

```
public static class TextClass
{
    public static string GetText()
    {
        return "message";
    }
}
```

Файл `TextClass.cs` с исходным кодом класса `TextClass` скомпилируем в виде модуля (обратите внимание на ключ компилятора):

```
csc.exe /t:module TextClass.cs
```

После компиляции получим программный модуль `TextClass.netmodule`. Далее, создадим консольное приложение (файл `MainClass.cs`):

[1] Visual Studio не позволяет работать с многофайловыми сборками, поэтому файлы примера нужно компилировать, используя компилятор командной строки `csc.exe`.

```
using System;

public class MainClass
{
    public static void Main()
    {
        Console.Write("Text from netmodule: ");
        Console.WriteLine(TextClass.GetText());
    }
}
```

Соберём многофайловую сборку. Ключ компилятора `/addmodule` позволяет добавить к сборке ссылку на внешний модуль. Этот ключ должен применяться для каждого подключаемого модуля.

```
csc.exe /addmodule:textclass.netmodule MainClass.cs
```

В итоге получим многофайловую сборку, состоящую из двух файлов: главного файла `mainclass.exe` и файла-модуля `textclass.netmodule`. Теперь можно создать новую сборку, в которой используется код из модуля `textclass.netmodule`, то есть сделать модуль *разделяемым* между несколькими сборками. Важное замечание: предполагается, что все файлы, составляющие нашу многофайловую сборку, размещены в одном каталоге.

Разберём вопрос взаимодействия сборок. Как правило, крупные программные проекты состоят из нескольких сборок, связанных ссылками. Среди этих сборок имеется некая основная (обычно оформленная как исполняемый файл), а другие сборки играют роль подключаемых библиотек с кодом необходимых типов (как правило, имеют расширение `*.dll`).

Представим пример, который будет использоваться в дальнейшем. Пусть имеется класс (в файле `UL.cs`), содержащий «полезную» функцию:

```
namespace UsefulLibrary
{
    public class UsefulClass
    {
        public void Print()
        {
            System.Console.WriteLine("Useful function");
        }
    }
}
```

Скомпилируем данный класс как библиотеку типов (расширение `*.dll`):

```
csc.exe /t:library UL.cs
```

Пусть основное приложение (файл `main.cs`) собирается использовать код из сборки `UL.dll`:

```
using System;
using UsefulLibrary;

public class MainClass
{
    public static void Main()
    {
        // используем класс из другой сборки
        UsefulClass a = new UsefulClass();
        a.Print();
    }
}
```

Ключ компилятора `/r` (или `/reference`) позволяет установить ссылку на требуемую сборку. Скомпилируем приложение `main.cs`:

```
csc.exe /r:UL.dll main.cs
```

В Visual Studio для добавления ссылок на сборки используется ветвь `References` в дереве проекта в Solution Explorer, или команда меню `Project | Add Reference`.

Создавая сборку, программист может снабдить её цифровой подписью, версией, указанием на региональную принадлежность и другой информацией. Эти данные внедряются в сборку при помощи атрибутов уровня сборки. Атрибуты можно разместить в любом файле с исходным кодом, необходимо только подключить пространство имён `System.Reflection`. Если для разработки используется Visual Studio, все атрибуты уровня сборки по умолчанию находятся в файле `AssemblyInfo.cs`. Кроме этого, в Visual Studio удобно редактировать атрибуты не напрямую, а используя диалоговое окно свойств проекта.

Цифровая подпись защищает сборку от несанкционированного изменения. Платформа .NET использует для цифровой подписи алгоритм RSA. Схема подписания сборки показана на рис. 8.

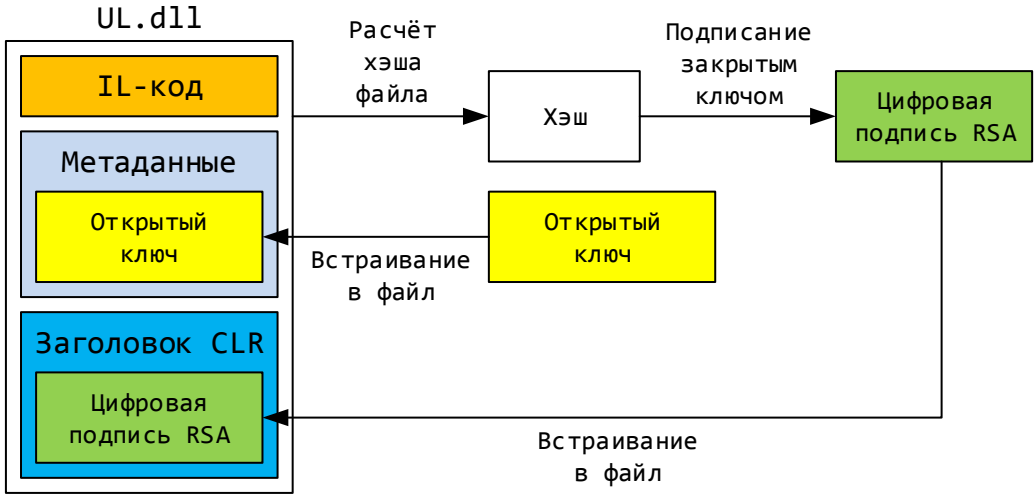

Рис. 8. Подписание сборки.

Для того чтобы снабдить сборку цифровой подписью, необходим *открытый ключ* и *закрытый ключ*. Утилита `sn.exe`, входящая в состав .NET Framework SDK, способна сгенерировать файл с парой ключей:

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

Чтобы подписать сборку, используется атрибут уровня сборки `[AssemblyKeyFile]` (имя файла с ключами нужно указать в качестве аргумента атрибута) или ключ компилятора командной строки `/keyfile`:

```
using System;
using System.Reflection;

[assembly: AssemblyKeyFile("keys.snk")]
namespace UsefulLibrary { . . . }
```

Сгенерировать ключи и подписать сборку можно в Visual Studio в окне свойств проекта (закладка `Signing`).

Чтобы указать версию сборки, используется атрибут `[AssemblyVersion]`. Номер версии имеет формат `Major.Minor.Build.Revision`. Часть `Major` является обязательной. Любая другая часть может быть опущена (в этом случае она полагается равной нулю). Часть `Revision` можно задать как *, тогда компилятор генерирует её как количество секунд, прошедших с полуночи, делённое на два. Часть `Build` также можно задать как *. Тогда для неё будет использовано количество дней, прошедших с 1 февраля 2000 года.

```
using System;
using System.Reflection;

[assembly: AssemblyVersion("1.2.0.0")]
namespace UsefulLibrary { . . . }
```

Для задания региональной принадлежности используется атрибут уровня сборки `[AssemblyCulture]`. Для определения версии и региональной принадлежности можно использовать окно свойств проекта в Visual Studio (закладка `Application`, кнопка `Assembly Information`).

Платформа .NET разделяет сборки на *локальные* (или *сборки со слабыми именами*) и *глобальные* (или *сборки с сильными именами*). Если `UL.dll` рассматривается как локальная сборка, то при выполнении приложения она должна находиться в том же каталоге, что и `main.exe`[1]. Локальные сборки обеспечивают простоту развёртывания приложения (все его компоненты сосредоточены в одном месте) и изолированность компонентов. Имя локальной сборки – *слабое имя* – это имя файла сборки без расширения.

[1] Технология *зондирования* (probing) позволяет размещать зависимые сборки в подкаталогах.

Хотя использование локальных сборок имеет свои преимущества, иногда необходимо сделать сборку общедоступной. До появления платформы .NET доминировал подход, при котором код общих библиотек помещался в системный каталог простым копированием файлов при установке. Такой подход привёл к проблеме, известной как *«ад DLL»* (DLL Hell). Инсталлируемое приложение могло заменить общую библиотеку новой версией, при этом другие приложения, ориентированные на старую версию библиотеки, переставали работать. Для устранения «ада DLL» в платформе .NET используется специальное защищённое *хранилище сборок* (Global Assembly Cache, GAC).

Сборки, помещаемые в GAC, должны удовлетворять определённым условиям. Во-первых, такие глобальные сборки должны иметь цифровую подпись. Это исключает подмену сборок злоумышленниками. Во-вторых, для глобальных сборок отслеживаются версии и языковые культуры. Допустимой является ситуация, когда в GAC находятся разные версии одной и той же сборки, используемые разными приложениями.

Сборка, помещённая в GAC, получает *сильное имя*. Как раз использование сильного имени является тем признаком, по которому среда исполнения понимает, что речь идёт не о локальной сборке, а о сборке из GAC. Сильное имя включает имя главного файла сборки (без расширения), версию сборки, указание о региональной принадлежности и маркер открытого ключа сборки:

Рассмотрим процесс создания сборки с сильным именем на примере сборки `UL.dll`. Во-первых, сборку необходимо снабдить цифровой подписью. После подписания для сборки можно указать версию и региональную принадлежность (это не обязательные действия). После этого сборку можно поместить в GAC. Простейший вариант – использовать утилиту `gacutil.exe`, входящую в состав .NET Framework SDK. При использовании ключа `/i` сборка помещается в GAC, а ключ `/u` удаляет сборку из GAC:

```
gacutil.exe /i UL.dll
```

Теперь сборка `UL.dll` помещена в GAC. Её сильное имя (для ссылки в программах) имеет вид:

```
UL, Version=1.2.0.0, Culture=neutral, PublicKeyToken=ff824814c57facfe
```

24. Метаданные и получение информации о типах

При создании сборки в неё помещаются метаданные, которые являются описанием всех типов в сборке и их элементов. Программист может работать с метаданными, используя специальный механизм, называемый *отражением* (reflection). Главные элементы, которые необходимы для использования возможностей отражения – это класс `System.Type`[1] и типы из пространств имён `System.Reflection` и `System.Reflection.Emit`.

[1] Профиль приложений Metro скрывает большинство элементов класса `Type` и открывает их в классе `System.Reflection.TypeInfo`.

Класс `Type` служит для хранения информации о типе. Существует несколько способов получить объект этого класса:

1. Вызвать у объекта метод `GetType()`. Данный метод определён на уровне `System.Object`, а значит, присутствует у любого объекта:

```
Foo foo = new Foo();         // Foo – это некий класс
Type t = foo.GetType();
```

2. Использовать статический метод `Type.GetType()`, которому передаётся имя типа в виде строки. Это имя должно быть полным, то есть включать пространство имён и, при необходимости, имя сборки, содержащей тип:

```
Type t1 = Type.GetType("System.Int32");
Type t2 = Type.GetType("SomeNamespace.Foo, MyAssembly");
```

3. Использовать операцию C# `typeof`, аргументом которой является тип:

```
Type t = typeof (Foo);
```

Операцию `typeof` можно применять к массивам и универсальным шаблонам. Причём в последнем случае допускается использовать как сконструированный тип, так и исходный тип-шаблон (обратите внимание на синтаксис записи универсального шаблона).

```
Type t1 = typeof (int[]);
Type t2 = typeof (char[,]);
Type t3 = typeof (List<int>);    // сконструированный тип
Type t4 = typeof (List<>);       // универсальный тип
```

Свойства класса `Type` позволяют узнать имя типа, имя базового типа, является ли тип универсальным, в какой сборке он размещается и другую информацию. Кроме этого, `Type` имеет специальные методы, возвращающие данные о полях типа, свойствах, событиях, методах и их параметрах.

Рассмотрим пример получения информации о типе. Будем анализировать примитивный тип `System.Int32`:

```
Type t = typeof (Int32);
Console.WriteLine("Full name = " + t.FullName);
Console.WriteLine("Base type is = " + t.BaseType);
Console.WriteLine("Is sealed = " + t.IsSealed);
Console.WriteLine("Is class = " + t.IsClass);
foreach (Type iType in t.GetInterfaces())
{
    Console.WriteLine(iType.Name);
}
foreach (FieldInfo fi in t.GetFields())
{
    Console.WriteLine("Field = " + fi.Name);
}
```

```
foreach (PropertyInfo pi in t.GetProperties())
{
    Console.WriteLine("Property = " + pi.Name);
}
foreach (MethodInfo mi in t.GetMethods())
{
    Console.WriteLine("Method Name = " + mi.Name);
    Console.WriteLine("Method Return Type = " + mi.ReturnType);
    foreach (ParameterInfo pr in mi.GetParameters())
    {
        Console.WriteLine("Parameter Name = " + pr.Name);
        Console.WriteLine("Type = " + pr.ParameterType);
    }
}
```

Как показывает пример, информация об элементах типа хранится в объектах классов `FieldInfo`, `PropertyInfo`, `MethodInfo` и т. п. Эти классы находятся в пространстве имён `System.Reflection`. Их иерархия показана на рис. 9.

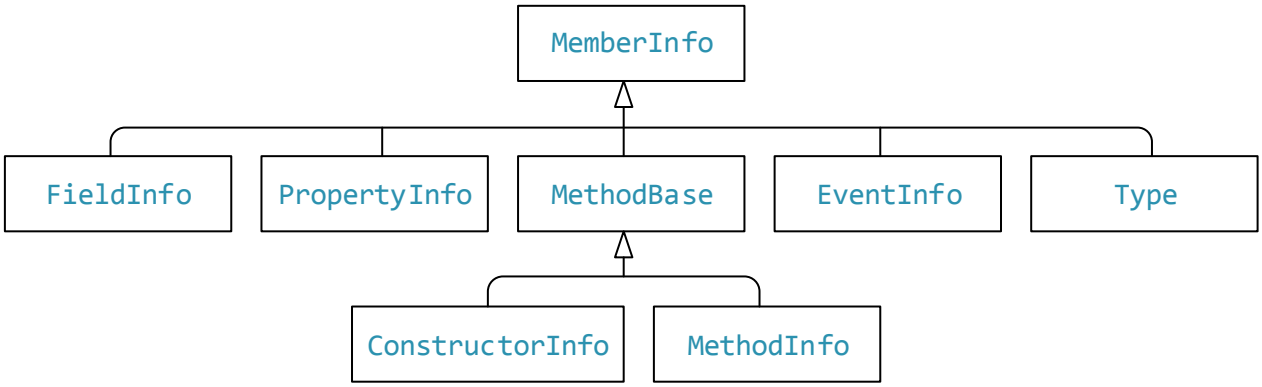

Рис. 9. Иерархия классов для представления элементов типа.

Отметим, что код примера покажет данные только об открытых элементах типа. Составом получаемой информации можно управлять, передавая в Get-методы дополнительные флаги перечисления `System.Reflection.BindingFlags` (табл. 16).

Таблица 16

Флаги `BindingFlags`, связанные с получением информации о типе

Флаг	Описание
`Default`	Отсутствие специальных флагов
`IgnoreCase`	Игнорировать регистр имён получаемых элементов
`DeclaredOnly`	Получить элементы, объявленные непосредственно в типе (игнорировать унаследованные элементы)
`Instance`	Получить экземплярные элементы
`Static`	Получить статические элементы
`Public`	Получить открытые элементы
`NonPublic`	Получить закрытые элементы
`FlattenHierarchy`	Получить `public` и `protected` элементы у типа и у всех его предков

```
Type t = typeof (Int32);
var bf = BindingFlags.Public | BindingFlags.NonPublic |
         BindingFlags.Static | BindingFlags.Instance;
FieldInfo[] fi = t.GetFields(bf);
```

Пространство имён `System.Reflection` содержит типы для получения информации и манипулирования сборкой и модулем сборки. При помощи класса `Assembly` можно получить информацию о сборке, при помощи класса `Module` – о модуле. Основные элементы этих классов перечислены в табл. 17 и табл. 18.

Таблица 17

Основные элементы класса `Assembly`

Имя элемента	Описание
`CreateInstance()`	Находит по имени тип в сборке и создаёт его экземпляр
`FullName`	Строковое свойство с полным именем сборки
`GetAssembly()`	Ищет в памяти и возвращает объект `Assembly`, который содержит указанный тип (статический метод)
`GetCustomAttributes()`	Получает атрибуты сборки
`GetExecutingAssembly()`	Возвращает сборку, которая содержит выполняемый в текущий момент код (статический метод)
`GetExportedTypes()`	Возвращает `public`-типы, определённые в сборке
`GetFiles()`	Возвращает файлы, из которых состоит сборка
`GetLoadedModules()`	Возвращает все загруженные в память модули сборки
`GetModule()`	Получает указанный модуль сборки
`GetModules()`	Возвращает все модули, являющиеся частью сборки
`GetName()`	Возвращает объект `AssemblyName` для сборки
`GetReferencedAssemblies()`	Возвращает объекты `AssemblyName` для всех сборок, на которые ссылается данная сборка
`GetTypes()`	Возвращает типы, определённые в сборке
`Load()`	Статический метод, который загружает сборку по имени
`LoadFrom()`	Статический метод; загружает сборку из указанного файла
`LoadModule()`	Загружает внутренний модуль сборки в память

Таблица 18

Основные элементы класса `Module`

Имя элемента	Описание
`Assembly`	Свойство с указанием на сборку (объект `Assembly`) модуля
`FindTypes()`	Получает массив классов, удовлетворяющих заданному фильтру
`FullyQualifiedName`	Строка, содержащая полное имя и путь к модулю
`GetType()`	Пытается выполнить поиск указанного типа в модуле
`GetTypes()`	Возвращает все типы, определённые в модуле
`Name`	Строка с коротким именем модуля

Продемонстрируем пример работы с классами `Assembly` и `Module`:

```
Assembly assembly = Assembly.GetExecutingAssembly();
Console.WriteLine(assembly.FullName);
foreach (Module module in assembly.GetModules())
{
    Console.WriteLine(module.FullyQualifiedName);
```

```
        foreach (Type type in module.GetTypes())
        {
            Console.WriteLine(type.FullName);
        }
    }
```

25. Позднее связывание и кодогенерация

Механизм отражения позволяет реализовать на платформе .NET *позднее связывание* (late binding). Этот термин обозначает процесс динамической загрузки сборок и типов при работе приложения, создание экземпляров типов и работу с их элементами.

Использование позднего связывания разберём на следующем примере. Пусть существует класс для хранения информации о человеке, и эта информация включает уникальный числовой идентификатор:

```
namespace Domain
{
    public class Person
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

Пусть имеется интерфейс, который описывает операции чтения и записи данных определённого человека из некоего хранилища:

```
namespace Domain
{
    public interface IRepository
    {
        Person Load(int id);
        void Save(Person person);
    }
}
```

Класс `Person` и интерфейс `IRepository` размещаются в сборке `Domain.dll`. Создадим сборку `Data.dll`, ссылающуюся на `Domain.dll` и предоставляющую реализацию интерфейса `IRepository`.

```
using System.IO;
using System.Runtime.Serialization;
using Domain;

namespace Data
{
    public class XmlRepository : IRepository
    {
        public Person Load(int id)
```

```
        {
            var ds = new DataContractSerializer(typeof (Person));
            using (Stream s = File.OpenRead(CreateName(id)))
            {
                return (Person) ds.ReadObject(s);
            }
        }

        public void Save(Person person)
        {
            var ds = new DataContractSerializer(typeof (Person));
            using (Stream s = File.OpenWrite(CreateName(person.Id)))
            {
                ds.WriteObject(s, person);
            }
        }

        private string CreateName(int id)
        {
            return id.ToString() + ".xml";
        }
    }
}
```

Теперь создадим консольное приложение `MainApp.exe`, которое записывает и читает объекты `Person`. Этот приложение ссылается на сборку `Domain.dll`, а со сборкой `Data.dll` будет работать опосредованно, без прямых ссылок. Такая архитектура позволит подменять механизм сохранения объектов `Person`, заменяя сборку `Data.dll` другой реализацией интерфейса `IRepository`.

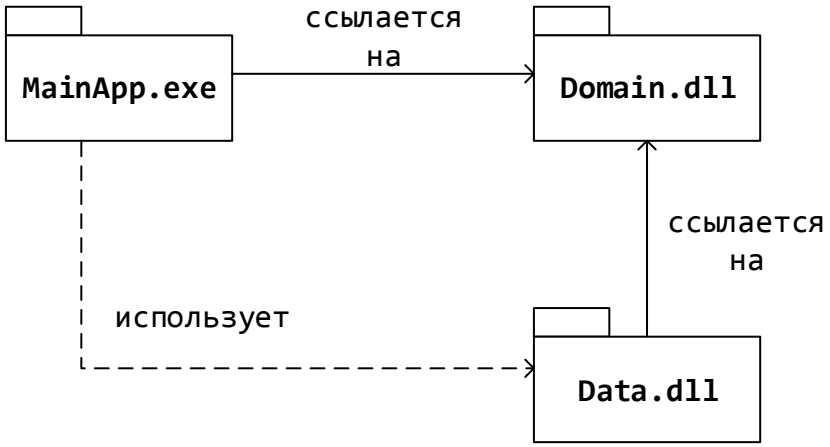

Рис. 10. Архитектура примера для работы с объектами `Person`.

Применим позднее связывание для работы с типами из сборки `Data.dll`. Первый этап позднего связывания – загрузка в память сборки с типом – выполняется при помощи метода `Assembly.Load()`. Для указания имени сборки можно использовать простую строку или объект класса `AssemblyName`.

```
// имя сборки обычно хранят в файле конфигурации
var assemblyName = new AssemblyName("Data");
try
{
    Assembly assembly = Assembly.Load(assemblyName);
    // здесь поместим код создания объекта
}
catch (FileNotFoundException)
{
    Console.WriteLine("Data.dll was not found");
}
```

После загрузки сборки создадим объект требуемого типа. Для этого можно воспользоваться статическим методом `Activator.CreateInstance()` или экземплярным методом класса `Assembly` с тем же именем:

```
// имя типа обычно хранят в файле конфигурации
Type type = assembly.GetType("Data.XmlRepository");
var repository = (IRepository) Activator.CreateInstance(type);
```

Методы `CreateInstance()` имеют множество перегрузок. Например, существует версия метода, принимающая массив объектов – аргументы конструктора создаваемого типа.

В нашем примере известно, что созданный объект реализует интерфейс `IRepository`, поэтому можно вызывать методы объекта обычным способом:

```
var person = new Person {Id = 12, Name = "John Doe"};
repository.Save(person);
Person clone = repository.Load(12);
```

Если при создании приложения нет информации об интерфейсе объекта, методы объекта можно вызвать альтернативными способами. Например, класс `MethodInfo` содержит экземплярный метод `Invoke()`. Его аргументы – целевой объект вызова и массив аргументов метода:

```
// модификация двух предыдущих фрагментов кода
Type type = assembly.GetType("Data.XmlRepository");
object repository = Activator.CreateInstance(type);

MethodInfo mi = type.GetMethod("Load");
var person = (Person) mi.Invoke(repository, new object[] {12});
```

Вызов метода при помощи отражения – медленная операция. Если так планируется вызывать метод несколько раз, выгоднее создать объект делегата, инкапсулирующий метод:

```
MethodInfo mi = type.GetMethod("Load");
var load = (Func<int, Person>) Delegate.CreateDelegate(
                typeof (Func<int, Person>), // тип делегата
```

```
                repository,                 // целевой объект
                mi);                        // метаинформация метода
Person person = load(12);
```

Механизм отражения позволяет не только исследовать готовые типы и выполнять для них позднее связывание, но и динамически создавать типы. Платформа .NET имеет средства генерации метаданных и инструкций языка CIL, сосредоточенные в пространстве имён `System.Reflection.Emit`. Рассмотрим простой пример – создание сборки и класса с единственным методом[1].

```
// 1. Создадим сборку
// 1a. Для этого получим домен (о них будет рассказано позднее)
AppDomain domain = AppDomain.CurrentDomain;

// 1b. Сформируем имя сборки
AssemblyName name = new AssemblyName("Library");

// 1c. Получаем сборку, которую затем собираемся сохранять
AssemblyBuilder ab = domain.DefineDynamicAssembly(name,
                            AssemblyBuilderAccess.RunAndSave);

// 2. В новой сборке определим модуль
ModuleBuilder mb = ab.DefineDynamicModule("Main", "Library.dll");

// 3. В модуле создадим класс Widget с уровнем доступа public
TypeBuilder tb = mb.DefineType("Widget", TypeAttributes.Public);

// 4. В класс добавим метод SayHello() без параметров
MethodBuilder method = tb.DefineMethod("SayHello",
                                       MethodAttributes.Public,
                                       null, null);

// 5. Формируем тело метода при помощи инструкций CIL
ILGenerator gen = method.GetILGenerator();
gen.EmitWriteLine("Hello world");
gen.Emit(OpCodes.Ret);

// 6. Завершаем создание класса
Type t = tb.CreateType();

// 7. Сохраняем полученную сборку
ab.Save("Library.dll");

// 8. Работаем с нашим классом, используя позднее связывание
var o = Activator.CreateInstance(t);
t.GetMethod("SayHello").Invoke(o, null); // Hello world
```

[1] Для генерации отдельных методов можно использовать класс `DynamicMethod`.

Ещё одно средство динамического создания кода предоставляют *деревья выражений*. Они описывают код в виде древовидной структуры. Каждый узел в дереве представляет выражение (например, вызов метода или бинарную операцию) и является объектом класса, унаследованного от `Expression` (пространство имён `System.Linq.Expressions`). Рис 11 демонстрирует некоторые классы-выражения.

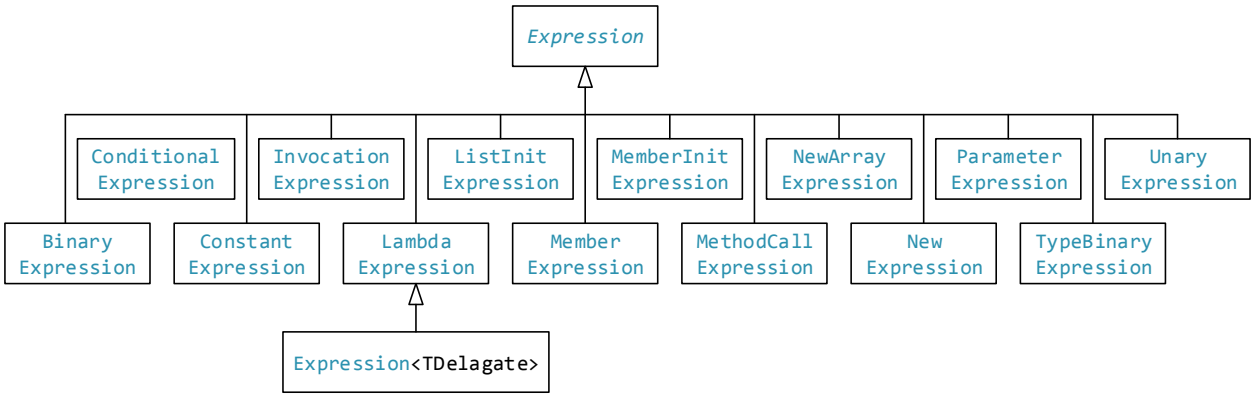

Рис. 11. Классы, унаследованные от `Expression`.

Универсальный класс `Expression<T>` позволяет создать дерево выражений на основе лямбда-выражения:

```
Func<int, bool> lambda = n => n < 5;              // обычная лямбда
Expression<Func<int, bool>> tree = n => n < 5;    // дерево
```

Класс `Expression` содержит статические методы, которые конструируют узлы дерева выражений особых типов:

```
// построим вручную дерево выражений для лямбды n => n < 5
ParameterExpression n = Expression.Parameter(typeof (int), "n");
ConstantExpression five = Expression.Constant(5, typeof (int));
BinaryExpression compare = Expression.LessThan(n, five);
Expression<Func<int, bool>> tree =
            Expression.Lambda<Func<int, bool>>(compare, new[] {n});
```

У созданного дерева выражений можно исследовать структуру, изменять элементы и компилировать его в инструкции CIL:

```
Expression<Func<int, bool>> tree = n => n < 5;
// декомпозиция дерева выражений
var param = tree.Parameters[0];
var op = (BinaryExpression) tree.Body;
var left = (ParameterExpression) op.Left;
var right = (ConstantExpression) op.Right;
Console.WriteLine("{0} => {1} {2} {3}",
                  param.Name, left.Name, op.NodeType, right.Value);
// компиляция дерева и вызов лямбды
Func<int, bool> lambda = tree.Compile();
Console.WriteLine(lambda(10));
```

26. Атрибуты

Платформа .NET позволяет расширять метаданные типов и сборок, используя систему атрибутов. Каждый *атрибут* (attribute) описывает дополнительную информацию, сохраняемую в метаданных для *цели атрибута* (attribute target) – сборки, модуля, типа, элементов типа, параметров метода. С точки зрения программиста атрибуту соответствует *класса атрибута* – это класс, наследуемый от `System.Attribute`.

Рассмотрим основные аспекты работы с атрибутами. Для создания атрибута нужно написать класс, удовлетворяющий перечисленным ниже требованиям:

1. Класс должен прямо или косвенно наследоваться от класса `Attribute`.

2. Тип открытых полей, свойств и параметров конструктора класса ограничен следующим набором: числовые типы (кроме `decimal`), `bool`, `char`, `string`, `object`, `System.Type`, перечисления; одномерные массивы указанных типов.

3. Имя класса должно заканчиваться суффиксом `Attribute` (это необязательное требование).

Приведём пример класса атрибута:

```
public class AuthorAttribute : Attribute
{
    public string Name { get; private set; }
    public string CreationDate { get; set; }

    public AuthorAttribute(string name)
    {
        Name = name;
    }
}
```

Для применения атрибута язык C# поддерживает следующий синтаксис: имя класса атрибута записывается в квадратных скобках перед тем элементом, к которому он относится. При этом разрешено указывать имя атрибута без суффикса `Attribute`. Применение атрибута условно соответствует созданию объекта. Поэтому после имени атрибута указываются в круглых скобках аргументы конструктора атрибута. Если у атрибута конструктор без параметров, круглые скобки можно не писать. Наряду с аргументами конструктора можно указать *именованные параметры*, предназначенные для задания значения открытого поля или свойства. Для этого используется синтаксис *`имя-элемента = значение`*. Именованные параметры всегда записываются в конце списка аргументов конструктора. Ниже приведены примеры применения `AuthorAttribute`.

```
[Author("Brian Kernighan", CreationDate = "01.01.2012")]
public class ColorPlugin
{
    [Author("Dennis Ritchie")]
    public void Process() { }
}
```

Для настройки создаваемого пользовательского атрибута можно использовать атрибут [AttributeUsage]. Конструктор [AttributeUsage] имеет единственный параметр – набор элементов перечисления AttributeTargets, определяющих цель атрибута. Булево свойство AllowMultiple определяет, может ли атрибут быть применён к программному элементу более одного раза. Булево свойство Inherited указывает, будет ли атрибут проецироваться на потомков программного элемента (по умолчанию значение свойства равно true).

```
// атрибут Author можно многократно применить к классу или методу
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
                AllowMultiple = true)]
public class AuthorAttribute : Attribute
{
    // элементы класса для краткости не приводятся
}
```

Синтаксис применения атрибутов позволяет указать в квадратных скобках несколько атрибутов через запятую. Если возникает неоднозначность трактовки цели атрибута, то нужно указать перед именем атрибута специальный префикс – assembly, module, field, event, method, param, property, return, type.

```
// применение атрибута к сборке
[assembly: AssemblyKeyFile("keys.snk")]

// многократное применение атрибута
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
public class ColorPlugin { }
```

Опишем возможности получения информации о применённых атрибутах. Метод Attribute.GetCustomAttributes() возвращает все атрибуты некоторого элемента в виде массива. Метод Attribute.GetCustomAttribute() получает атрибут заданного типа:

```
Attribute GetCustomAttribute(MemberInfo element, Type attributeType)
```

При помощи параметра element задаётся элемент, у которого надо получить атрибут. Второй параметр – это тип получаемого атрибута.

```
// пример получения атрибута
var author = Attribute.GetCustomAttribute(typeof (ColorPlugin),
                                          typeof (AuthorAttribute));
if (author != null)
{
    Console.WriteLine(((AuthorAttribute) author).Name);
}
```

Платформа .NET предоставляет для использования обширный набор атрибутов, некоторая часть которых представлена в табл. 19.

Таблица 19

Некоторые атрибуты, применяемые в платформе .NET

Атрибут	Цель применения	Описание
[AttributeUsage]	Класс	Задаёт область применения класса-атрибута
[CallerFilePath], [CallerLineNumber], [CallerMemberName]	Опциональный параметр	Позволяют получить в качестве значений опциональных параметров информацию о месте вызова метода в исходном коде программы
[Conditional]	Метод	Компилятор может игнорировать вызовы помеченного метода при заданном условии
[DllImport]	Метод	Импорт функций из DLL
[MTAThread]	Метод Main()	Для приложения используется модель COM Multithreaded apartment
[NonSerialized]	Поле	Указывает, что поле не будет сериализовано
[Obsolete]	Кроме param, assembly, module, return	Информирует, что в будущих реализациях данный элемент может отсутствовать
[ParamArray]	Параметр	Позволяет одиночному параметру быть обработанным как набор параметров params
[Serializable]	Класс, структура, перечисление, делегат	Указывает, что все поля типа могут быть сериализованы
[STAThread]	Метод Main()	Для приложения используется модель COM Single-threaded apartment
[StructLayout]	Класс, структура	Задаёт схему размещения данных класса или структуры в памяти (Auto, Explicit, Sequential)
[ThreadStatic]	Статическое поле	В каждом потоке будет использоваться собственная копия данного статического поля

Рассмотрим единичный пример использования стандартных атрибутов. Атрибуты применяются для настройки взаимодействия программ платформы .NET и библиотек на неуправляемом коде. Атрибут [DllImport] предназначен для импортирования функций из DLL, написанных на неуправляемом коде. В следующей программе показан импорт системной функции MessageBoxA():

```
using System.Runtime.InteropServices;

public class MainClass
{
    [DllImport("user32.dll")]
    public static extern int MessageBoxA(int hWnd, string text,
                                         string caption, uint type);

    public static void Main()
    {
        MessageBoxA(0, "Hello World", "Unmanaged DLL", 0);
    }
}
```

Для использования атрибута [DllImport] требуется подключить пространство имён System.Runtime.InteropServices. Кроме этого, необходимо объявить импортируемую функцию статической и пометить её модификатором extern. Атрибут [DllImport] допускает использование дополнительных аргументов, подробное описание которых можно найти в документации MSDN.

Исполняемая среда .NET выполняет корректную передачу аргументов примитивных типов между управляемым и неуправляемым кодом. Для правильной передачи сложных аргументов требуется использование специального атрибута [StructLayout] при объявлении пользовательского типа. Например, пусть выполняется экспорт системной функции GetLocalTime():

```
[DllImport("kernel32.dll")]
public static extern void GetLocalTime(SystemTime st);
```

В качестве параметра функция использует объект класса SystemTime. Этот класс должен быть описан следующим образом:

```
[StructLayout(LayoutKind.Sequential)]
public class SystemTime
{
    public ushort wYear;
    public ushort wMonth;
    public ushort wDayOfWeek;
    public ushort wDay;
    public ushort wHour;
    public ushort wMinute;
    public ushort wSecond;
    public ushort wMilliseconds;
}
```

Атрибут [StructLayout] с аргументом LayoutKind.Sequential указывает, что поля объекта должны быть расположены в памяти в точности так, как это записано в объявлении класса. В противном случае при работе с системной функцией вероятно возникновение ошибок.

27. Динамическое связывание

Определение значения операции, основывающееся на типе или значении составных частей выражения (аргументов, операндов, получателей) в языке C# обычно происходит на этапе компиляции и именуется *статическим связыванием* (static binding). Если этот процесс осуществляется при выполнении программы, то он называется *динамическим связыванием* (dynamic binding). Язык C# поддерживает динамическое связывание, начиная с четвёртой версии. Цель динамического связывания – позволить программам на C# взаимодействовать с *динамическими объектами*, то есть с объектами, которые не подчиняются обычным правилам системы типов C#. Примерами таких объектов являются:

– объекты динамических языков (IronPython, IronRuby, Jscript);

– «расширяемые» объекты, которые позволяют добавлять новые свойства во время выполнения программы (Internet Explorer DOM);

– объекты COM (например, в объектной модели Microsoft Office).

Для указания на необходимость динамического связывания в языке C# используется особый тип – `dynamic`. У объекта такого типа можно записать вызов любого метода или свойства, это не влияет на компиляцию.

```
public class Foo
{
    public void Print(string s)
    {
        Console.WriteLine(s);
    }
}

// этот код компилируется,
// но при выполнении третья строка генерирует исключение
dynamic obj = new Foo();
obj.Print("Hello world");
obj.Property = 10;    // RuntimeBinderException
```

Тип `dynamic` может использоваться при объявлении локальной переменной, формального параметра или возвращаемого значения метода, элемента типа (поля, свойства), сконструированного универсального шаблона. Определено неявное преобразование `dynamic` и других типов друг в друга.

```
// метод с dynamic-параметрами и возвращаемым значением
public dynamic Div(dynamic x, dynamic y)
{
    return x/y;
}

// неявные преобразования
int i = 7;
dynamic d = i;
int j = d;
```

В качестве примера использования `dynamic` рассмотрим класс, реализующий *перечисляемый слаботипизированный кортеж*:

```
public class IterableTuple : IEnumerable
{
    private readonly List<dynamic> _storage;

    public IterableTuple(params dynamic[] args)
    {
        _storage = new List<dynamic>(args);
    }
```

```
    public static IterableTuple Create(params dynamic[] args)
    {
        return new IterableTuple(args);
    }

    public dynamic this[int i]
    {
        get
        {
            return _storage[i];
        }
    }

    public IEnumerator GetEnumerator()
    {
        return _storage.GetEnumerator();
    }
}

// пример использования IterableTuple
var tuple = IterableTuple.Create(1.5, 12, "string", 69);
foreach (var item in tuple)
{
    Console.WriteLine(item);
}
```

Компилятор заменяет объявление `dynamic` на объявление `object`, поэтому с точки зрения CLR эти типы эквиваленты. Работу с `dynamic`-объектом компилятор организует, используя класс `Microsoft.CSharp.RuntimeBinder.Binder` (сборка `Microsoft.CSharp.dll`). Способ нахождения элементов динамического объекта зависит от его конкретного типа:

1. Обычные объекты .NET – элементы определяются при помощи механизма отражения, работа с элементами происходит при помощи позднего связывания.

2. Объект, реализующий интерфейс `IDynamicMetaObjectProvider`, – сам объект запрашивается о том, содержит ли он заданный элемент. В случае успеха работа с элементом делегируется объекту.

3. COM-объекты – работа происходит через интерфейс `IDispatch`.

Интерфейс `IDynamicMetaObjectProvider` позволяет разработчикам создавать типы, обладающие динамическим поведением. Обычно данный интерфейс не реализуется напрямую, а выполняется наследование от класса `DynamicObject` (интерфейс и класс находятся в пространстве имён `System.Dynamic`). В качестве примера использования `DynamicObject` приведём класс, обеспечивающий динамический доступ к атрибутам XML-элемента:

```
public static class XExtensions
{
    public static dynamic DynamicAttributes(this XElement e)
```

```
        {
            return new XWrapper(e);
        }

        private class XWrapper : DynamicObject
        {
            private readonly XElement _element;

            public XWrapper(XElement e)
            {
                _element = e;
            }

            // метод вызывается при попытке прочитать значение свойства
            public override bool TryGetMember(GetMemberBinder binder,
                                              out object result)
            {
                result = _element.Attribute(binder.Name).Value;
                return true;
            }

            // метод вызывается при попытке установить значение свойства
            public override bool TrySetMember(SetMemberBinder binder,
                                              object value)
            {
                _element.SetAttributeValue(binder.Name, value);
                return true;
            }
        }
    }

    // пример работы с XWrapper
    XElement x = XElement.Parse(@"<Label Text=""Hello"" Id=""5""/>");
    dynamic da = x.DynamicAttributes();
    Console.WriteLine(da.Id);           // 5
    da.Text = "Foo";
    Console.WriteLine(x.ToString());    // <Label Text="Foo" Id="5" />
```

Класс `System.Dynamic.ExpandoObject` позволяет при выполнении программы добавлять и удалять элементы своего экземпляра:

```
public sealed class ExpandoObject : IDynamicMetaObjectProvider,
                                    IDictionary<string, object>,
                                    INotifyPropertyChanged
```

Благодаря динамической типизации работа с пользовательскими элементами `ExpandoObject` происходит как работа с обычными элементами объекта. Ниже приведён пример расширения `ExpandoObject` двумя свойствами и методом (в виде делегата).

```
dynamic sample = new ExpandoObject();
sample.Caption = "The caption";     // добавляем свойство Caption
sample.Number = 10;                 // и числовое свойство Number
sample.Increment = (Action) (() => { sample.Number++; });

// работаем с объектом sample
Console.WriteLine(sample.Caption);              // The caption
Console.WriteLine(sample.Caption.GetType());    // System.String
sample.Increment();
Console.WriteLine(sample.Number);               // 11
```

Объект `ExpandoObject` явно реализует `IDictionary<string, object>`. Это позволяет инспектировать элементы объекта при выполнении программы и при необходимости удалять их.

```
// этот метод преобразует ExpandoObject в XElement
public static XElement ExpandoToXml(dynamic node, string nodeName)
{
    var xmlNode = new XElement(nodeName);
    foreach (var property in (IDictionary<string, object>) node)
    {
        if (property.Value is ExpandoObject)
        {
            xmlNode.Add(ExpandoToXml(property.Value, property.Key));
        }
        else if (property.Value.GetType() == typeof (List<dynamic>))
        {
            foreach (dynamic el in (List<dynamic>) property.Value)
            {
                xmlNode.Add(ExpandoToXml(el, property.Key));
            }
        }
        else
        {
            xmlNode.Add(new XElement(property.Key, property.Value));
        }
    }
    return xmlNode;
}

// пример работы с ExpandoToXml()
dynamic contact = new ExpandoObject();
contact.Name = "Patrick Hines";
contact.Phone = "206-555-0144";
contact.Address = new ExpandoObject();
contact.Address.Street = "123 Main St";
contact.Address.City = "Mercer Island";
contact.Address.State = "WA";
contact.Address.Postal = "68402";
```

```
dynamic res = ExpandoToXml(contact, "Contact");
Console.WriteLine(res);

// будет выведен следующий XML-фрагмент
// <Contact>
//   <Name>Patrick Hines</Name>
//   <Phone>206-555-0144</Phone>
//   <Address>
//     <Street>123 Main St</Street>
//     <City>Mercer Island</City>
//     <State>WA</State>
//     <Postal>68402</Postal>
//   </Address>
// </Contact>
```

28. Файлы конфигурации

Конфигурирование применяется для решения двух основных задач. Во-первых, параметры конфигурации позволяют настроить поведение CLR при выполнении кода приложения. Во-вторых, конфигурация может хранить пользовательские данные приложения.

Платформа .NET предлагает унифицированный подход к конфигурированию, основанный на использовании *конфигурационных XML-файлов*. Существует один глобальный файл конфигурации с параметрами, относящимися к платформе в целом. Этот файл называется `machine.config` и располагается в каталоге установки платформы .NET. Любая сборка может иметь локальный конфигурационный файл. Он должен носить имя файла сборки с добавлением расширения `.config` и располагаться в одном каталоге со сборкой (то есть файл конфигурации для `main.exe` должен называться `main.exe.config`[1]). Параметры, описанные в локальных конфигурационных файлах, «накладываются» на параметры из файла `machine.config`.

Проанализируем общую схему любого файла конфигурации. Корневым XML-элементом файла является элемент `<configuration>`. Он может включать следующие дочерние элементы:

`<configSections>` – описывает разделы конфигурации (в том числе пользовательские);

`<appSettings>` – пользовательские параметры конфигурации;

`<connectionStrings>` – строки подключения к источникам данных;

`<startup>` – параметры запуска CLR (поддерживаемые версии);

`<runtime>` – параметры времени выполнения (регулируют способ загрузки сборок и работу сборщика мусора);

`<system.diagnostics>` – совокупность диагностических параметров, которые задают способ отладки, перенаправляют сообщения отладки и т. д.;

`<system.net>` – настройка параметров работы с сетью;

[1] В случае веб-приложения файл конфигурации всегда называется `web.config`.

`<system.serviceModel>` – настройка элементов технологии WCF;

`<system.web>` – параметры конфигурации приложений ASP.NET.

Рассмотрим способы описания в файле конфигурации пользовательских данных. В простейшем случае для этого используется раздел `<appSettings>`, который может содержать следующие элементы:

`<add key="name" value="the value"/>` – добавляет новый ключ и значение в коллекцию пользовательских конфигурационных данных;

`<remove key="name"/>` – удаляет существующий ключ и значение из коллекции конфигурационных данных;

`<clear/>` – очищает коллекцию конфигурационных данных.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="company" value="Acme, Ltd"/>
    <add key="year" value="2012"/>
  </appSettings>
</configuration>
```

Разработчик может создать собственный раздел конфигурационного файла. Такой раздел должен быть зарегистрирован в секции `<configSections>`. При регистрации раздела задаётся его *обработчик* – класс, который будет отвечать за превращение содержимого раздела в данные. В зависимости от типа хранимых данных можно воспользоваться одним из существующих обработчиков либо построить собственный обработчик.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="custom" type="Common.CustomConfig, Common"/>
  </configSections>

  <!-- пользовательская секция -->
  <custom>
    <copyright company="Acme, Ltd" year="2012"/>
  </custom>
</configuration>
```

Если планируется использование собственного обработчика конфигурационного раздела, следует создать класс, производный от `ConfigurationSection`[1]. В классе определяются открытые свойства, соответствующие XML-атрибутам

[1] Подробную информацию о стандартных классах, используемых в механизме конфигурации, содержит серия статей по адресу codeproject.com/Articles/16466/Unraveling-the-Mysteries-of-NET-2-0-Configuration.

или вложенным элементам конфигурации. Этим свойствам назначается специальный атрибут [ConfigurationProperty]. Следующий код показывает пример пользовательского обработчика CustomConfig.

```
namespace Common
{
    public class CustomConfig : ConfigurationSection
    {
        [ConfigurationProperty("copyright", IsRequired = true)]
        public Element Copyright
        {
            get { return (Element) base["copyright"]; }
        }
    }

    public class Element : ConfigurationElement
    {
        [ConfigurationProperty("company", IsRequired = true)]
        public string Company
        {
            get { return (string) base["company"]; }
            set { base["company"] = value; }
        }

        [ConfigurationProperty("year", IsRequired = true)]
        public int Year
        {
            get { return (int) base["year"]; }
            set { base["year"] = value; }
        }
    }
}
```

Для программного доступа к конфигурационным данным текущего приложения используется статический класс ConfigurationManager из пространства имён System.Configuration[1] (сборка System.Configuration.dll). Класс имеет следующие элементы:

1. AppSettings – коллекция-словарь пользовательских параметров;
2. ConnectionStrings – словарь строк подключения к источникам данных;
3. GetSection() – извлекает указанный раздел конфигурации;
4. OpenExeConfiguration() – открывает файл конфигурации указанной сборки в виде объекта Configuration;
5. OpenMachineConfiguration() – открывает файл machine.config;
6. RefreshSection() – перечитывает указанный раздел конфигурации.

Покажем пример работы с коллекцией данных <appSettings> и пользовательским разделом конфигурации при помощи ConfigurationManager:

[1] В веб-приложениях используется System.Web.Configuration.WebConfigurationManager.

```
string name = ConfigurationManager.AppSettings["company"];

var s = (CustomConfig) ConfigurationManager.GetSection("custom");
string company = s.Copyright.Company;
int year = s.Copyright.Year;
Console.WriteLine("Copyright © {0} by {1}", year, company);
```

Метод `ConfigurationManager.OpenExeConfiguration()` позволяет загрузить конфигурацию заданной сборки в виде объекта `Configuration`. Это класс содержит коллекции, описывающие секции конфигурационного файла, а также методы для записи конфигурационного файла.

```
// получаем конфигурацию сборки common.exe
var cfg = ConfigurationManager.OpenExeConfiguration("common.exe");

// находим секцию и изменяем данные в ней
var section = (CustomConfig) cfg.Sections["custom"];
section.Copyright.Year = 2013;

// обновляем конфигурацию
cfg.Save();
```

При работе в Visual Studio можно создать так называемые *пользовательские настройки*. Для этого используется окно свойств проекта (`Project | Properties | Settings`). Для каждого параметра указывается имя, тип и область видимости – глобальная или локальная (для конкретного пользователя).

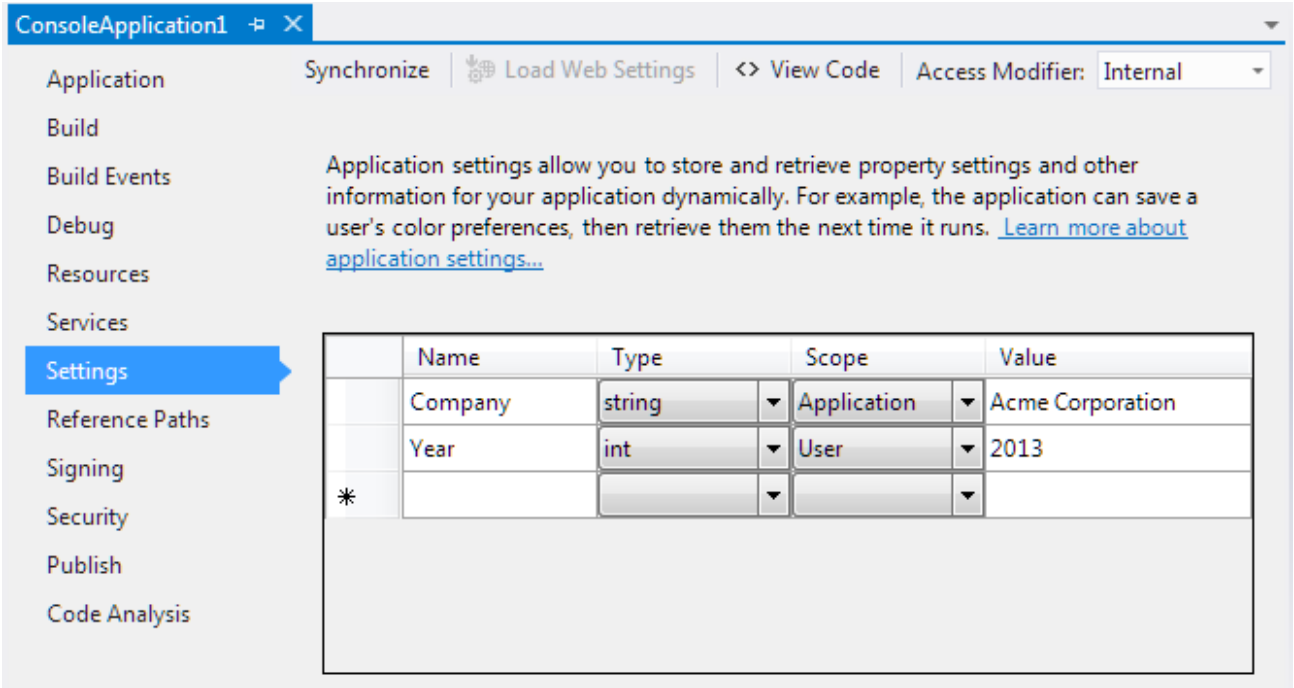

Рис. 12. Редактирование пользовательских настроек.

Visual Studio автоматически генерирует для работы с пользовательскими настройками класс `Settings`, размещённый в подпространстве имён `Properties`.

Параметры настройки доступны через свойство `Settings.Default`. Если у параметра локальная область видимости, его можно не только прочитать, но и изменить, а затем сохранить.

```
Console.WriteLine(Settings.Default.Company);
Console.WriteLine(Settings.Default.Year);
Settings.Default.Year = 2013;
Settings.Default.Save();
```

29. Диагностика и мониторинг

Рассмотрим некоторые средства языка C# и платформы .NET, облегчающие диагностику проблем и мониторинг поведения приложения.

Описание средств диагностики начнём с *условной компиляции*, выполняемой при помощи специальных директив препроцессора. Директива `#define` служит для перевода *символа условной компиляции* в состояние «определён». Символом условной компиляции может служить любой идентификатор или ключевое слово, за исключением `true` или `false`. Директива `#undef` переключает символ условной компиляции в состояние «не определён». Обе директивы должны располагаться в самом начале файла с исходным кодом и могут повторяться произвольное количество раз.

Директивы `#if` и `#endif` позволяют задать в исходном коде регион, который компилируется, только если символ условной компиляции, указанный после `#if`, определён.

```
#define TESTMODE
using System;

public class Program
{
    public static void Main()
    {
#if TESTMODE
        Console.WriteLine("In test mode!");
#endif
    }
}
```

Внутри региона `#if` – `#endif` можно применять директивы `#else`, `#elif` (это эквивалент `#else #if`), а символы условной компиляции после `#if` комбинировать при помощи операций `&&`, `||` и `!`.

```
#define TESTMODE
#define V2

using TestType =
#if V2 && TESTMODE
    MyCompany.GadgetV2;
```

```
#else
    MyCompany.Gadget;
#endif
```

Заметим, что символ условной компиляции можно определить не только с помощью `#define`, но и указав ключ компилятора командной строки `/define` или используя окно свойств проекта в Visual Studio (в этих случаях символ считается определённым не в отдельном файле, а во всех файлах проекта). Visual Studio назначает в отладочной конфигурации проекта символы условной компиляции `DEBUG` и `TRACE`, а в выпускной конфигурации – только символ `TRACE`.

Метод класса или структуры, который возвращает значение `void`, может быть помечен атрибутом `[Conditional]` с указанием символа условной компиляции. Если символ не определён, компилятор исключит из кода все вызовы помеченного метода.

```
[Conditional("TESTMODE")]
private void WriteInLog(string message)
{
    Console.WriteLine(message);
}
```

Атрибутом `[Conditional]` может быть помечен и класс атрибута. В этом случае помеченный атрибут применяется к своей цели, только если указанный символ условной компиляции определён.

При мониторинге работы программы полезным оказывается возможность вывода и фиксации различных диагностических сообщений. Для этой цели можно применить статические классы `Debug` и `Trace` из пространства имён `System.Diagnostics`. Эти классы похожи, но все методы класса `Debug` помечены атрибутом `[Conditional("DEBUG")]`, а все методы класса `Trace` – атрибутом `[Conditional("TRACE")]`. Классы `Debug` и `Trace` содержат статические методы вывода сообщений `Write()`, `WriteLine()`, `WriteIf()`, `Fail()` и `Assert()`. В классе `Trace` определены дополнительные методы `TraceInformation()`, `TraceWarning()` и `TraceError()`.

```
// примеры вызова методов у классов Debug и Trace
Debug.WriteLine("Debug message");
Debug.WriteIf(5 > 2, "Correct condition");
Debug.Fail("Error in code");
Debug.Assert(3 < 2, "Error in condition");
Trace.TraceInformation("Info");
```

Классы `Debug` и `Trace` имеют свойство `Listeners`, содержащее коллекцию *слушателей* – объектов классов, производных от `TraceListener`. Слушатели ответственны за обработку содержимого, генерируемого методами вывода сообщений. По умолчанию коллекция `Listeners` включает одного слушателя – объект класса `DefaultTraceListener`. Этот слушатель записывает сообщения в окно отладчика Visual Studio, а при вызове метода `Fail()` или нарушении условия в

методе Assert() выводит диалоговое окно для подтверждения выполнения программы. Данное поведение можно изменить, удалив слушатель или добавив один или несколько собственных слушателей. Слушатель можно создать, унаследовав от класса TraceListener или воспользоваться одним из готовых классов:

TextWriteTraceListener – пишет в поток, или в TextWriter, или в файл. Имеет подклассы ConsoleTraceListener, DelimitedListTraceListener, XmlWriterTraceListener, EventSchemaTraceListener.

EventLogTraceListener – пишет в журнал событий Windows. При этом сообщения, выводимые методами TraceWarning() и TraceError(), выводятся как предупреждения и ошибки соответственно.

EventProviderTraceListener – пишет в подсистему ETW в Windows Vista.

WebPageTraceListener – пишет на веб-страницу ASP.NET.

```
// удаляем слушатель по умолчанию
Trace.Listeners.Clear();

// добавляем слушатель для записи в конец файла
Trace.Listeners.Add(new TextWriterTraceListener("trace.txt"));

// настраиваем журнал событий Windows
if (!EventLog.SourceExists("DemoApp"))
{
    EventLog.CreateEventSource("DemoApp", "Application");
}

// и добавляем соответствующий слушатель
Trace.Listeners.Add(new EventLogTraceListener("DemoApp"));
// пишем несколько сообщений в файл и журнал событий
Trace.WriteLine("message");
Trace.TraceError("error");
```

В классе TraceListener определено свойство Filter, которое можно установить для фильтрации сообщений. Кроме этого, имеется несколько свойств для управления внешним видом выводимых сообщений.

```
var listener = new TextWriterTraceListener("trace.txt");
listener.Filter = new SourceFilter("Program");
listener.IndentSize = 4;
```

Для слушателей, которые пишут информацию в кэшируемый поток (например, слушатель TextWriteTraceListener), рекомендуется перед окончанием работы приложения вызывать методы Close() или Flush(). Такие же методы есть у классов Debug и Trace. Если у этих классов установить свойство AutoFlush в true, то метод Flush() будет вызываться после каждого сообщения.

Отметим, что настройка слушателей может быть выполнена не только программно, но и декларативно – с использованием конфигурационного файла приложения (секция <system.diagnostics>):

```
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="2">
      <listeners>
        <add name="lst"
             type="System.Diagnostics.TextWriterTraceListener,
                   System, Version=1.0.3300.0, Culture=neutral,
                   PublicKeyToken=b77a5c561934e089"
             initializeData="MyListener.log"
             traceOutputOptions="ProcessId, Timestamp" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

Ещё один полезный класс из пространства имён `System.Diagnostics` – класс `Stopwatch`. Он предоставляет набор методов и средств, которые можно использовать для точного измерения затраченного времени. В типовом сценарии использования у экземпляра `Stopwatch` вызывается метод `Start()`, затем – метод `Stop()`, а затраченное время проверяется при помощи свойства `Elapsed`.

```
// замерим скорость подсчёта факториала 10000
var sw = new Stopwatch();
sw.Start();
BigInteger factorial = 1;
for (int i = 2; i < 10000; i++)
{
    factorial *= i;
}
sw.Stop();
Console.WriteLine(sw.Elapsed);    // 00:00:00.0909346
```

30. Процессы и домены

Любому запущенному приложению в операционной системе соответствует некий *процесс*. Процесс образует границы приложения, выделяя для приложения изолированное адресное пространство и поддерживая один или несколько потоков выполнения. Для работы с процессами в платформе .NET имеется класс `System.Diagnostics.Process`. Используя статические и экземплярные элементы этого класса можно получить информацию о текущем процессе, а также обо всех процессах системы.

```
Process current = Process.GetCurrentProcess();
foreach (Process p in Process.GetProcesses())
{
    Console.WriteLine("{0} {1} {2}",
                      p.Id, p.ProcessName, p.StartTime);
}
```

Класс `Process` позволяет управлять процессами (при наличии соответствующих привилегий). В следующем примере запускается приложение «Блокнот», которое завершается через 5 секунд.

```
Process p = Process.Start("notepad.exe");
Thread.Sleep(5000);
p.Kill();
```

Платформа .NET вводит дополнительный уровень изоляции кода, называемый *доменом приложения*. Домены существуют внутри процессов и содержат загруженные сборки (можно создать сборку, разделяемую между доменами). Любой процесс запускает при старте домен по умолчанию, однако домены могут создаваться и уничтожаться в ходе работы в рамках процесса. Домены обеспечивают приемлемый уровень изоляции кода, но их создание менее затратное, чем создание отдельных процессов. Кроме того, домен можно уничтожить, не нарушая целостность работы всего процесса.

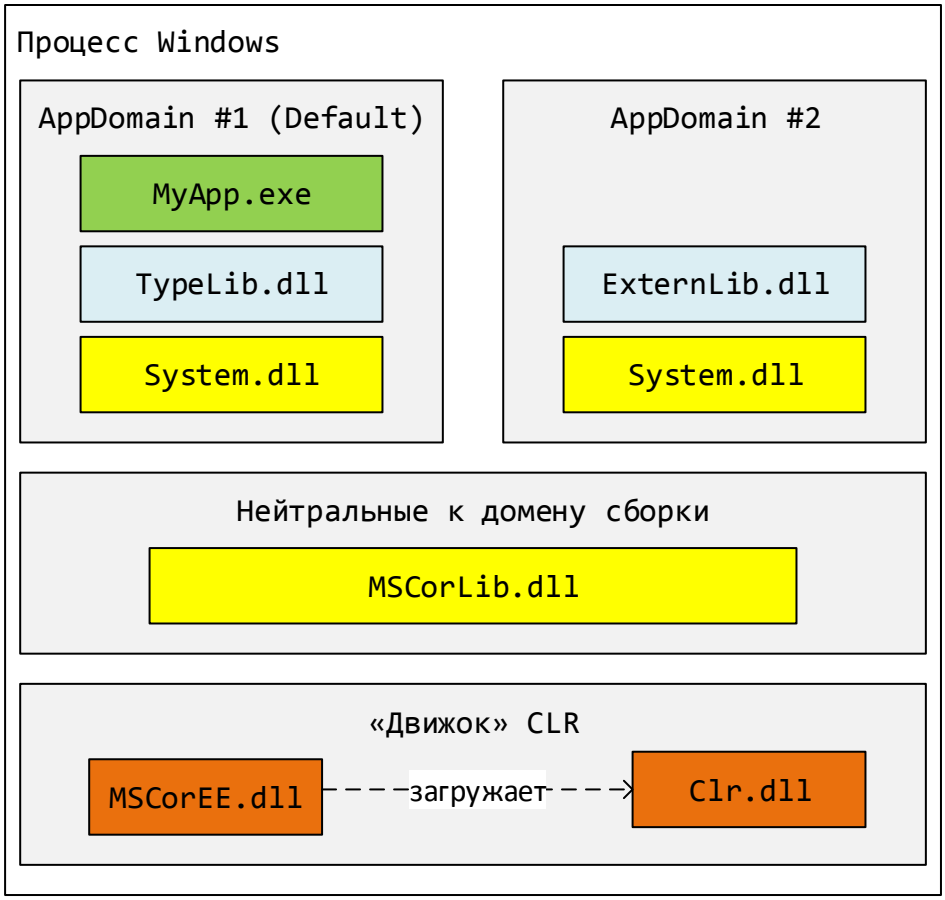

Рис. 13. Структура .NET-процесса с двумя доменами.

Домены приложений инкапсулированы в объектах класса `System.AppDomain`. Статическое свойство `CurrentDomain` позволяет получить информацию о текущем домене, а статические методы наподобие `CreateDomain()` – создать новый

домен в рамках текущего процесса. После создания домена в него можно программно загрузить сборки, используя экземплярный метод `Load()`. Выгрузить сборки из домена нельзя, можно выгрузить весь домен:

```
AppDomain newDomain = AppDomain.CreateDomain("New Domain");
newDomain.Load("assemblyName");
AppDomain.Unload(newDomain);
```

Событие `UnhandledException` объекта `AppDomain` генерируется при возникновении в домене необработанного исключения:

```
AppDomain domain = AppDomain.CurrentDomain;
domain.UnhandledException +=
          (sender, args) => Console.WriteLine(args.ExceptionObject);

// необработанное исключение генерирует событие UnhandledException
int x = 0;
int y = 10/x;
```

Домены содержат методы для создания экземпляров объектов требуемых типов (например, `CreateInstance()`). Однако доступ к созданным экземплярам нетривиален – фактически, это межпрограммное взаимодействие, для которого существуют специальные технологии.

31. Основы многопоточного программирования

Платформа .NET даёт полноценную поддержку для создания многопоточных приложений. Исполняющая среда имеет особый модуль, ответственный за организацию многопоточности, но в основном работа модуля опирается на функции многопоточности операционной системы. В этом параграфе рассматриваются базовые приёмы создания многопоточных приложений.

Основные классы, предназначенные для поддержки многопоточности, сосредоточены в пространстве имён `System.Threading`. На платформе .NET каждый *поток выполнения* (thread) представлен объектом класса `Thread`. Для организации собственного потока необходимо создать объект этого класса. Класс `Thread` имеет четыре перегруженные версии конструктора:

```
public Thread(ThreadStart start);
public Thread(ThreadStart start, int maxStackSize);
public Thread(ParameterizedThreadStart start);
public Thread(ParameterizedThreadStart start, int maxStackSize);
```

В качестве первого аргумента конструктору передаётся делегат, инкапсулирующий метод, выполняемый в потоке. Доступно два типа делегатов: второй позволяет при запуске метода передать ему данные в виде объекта:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);
```

Дополнительный параметр конструктора класса `Thread` может использоваться для указания максимального размера стека, выделяемого потоку[1].

Создание потока не подразумевает его автоматического запуска. Для запуска потока требуется вызвать метод `Start()` (перегруженная версия метода получает объект, передаваемый как аргумент методу потока).

```
var th = new Thread(DoSomeWork);
th.Start();
```

Рассмотрим основные свойства класса `Thread`:

1. Статическое свойство `CurrentThread` возвращает объект, представляющий текущий поток.

2. Свойство `Name` служит для назначения потоку имени.

3. Целочисленное свойство для чтения `ManagedThreadId` возвращает уникальный числовой идентификатор управляемого потока.

4. Свойство для чтения `ThreadState`, значением которого являются элементы одноимённого перечисления, позволяет получить текущее состояние потока.

5. Булево свойство для чтения `IsAlive` позволяет определить, выполняется ли поток.

6. Свойство `Priority` управляет приоритетом выполнения потока относительно текущего процесса. Значением этого свойства являются элементы перечисления `ThreadPriority`: `Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, `Highest`.

7. Булево свойство `IsBackground` позволяет сделать поток фоновым. Среда исполнения .NET разделяет все потоки на *фоновые* и *основные*. Процесс не может завершиться, пока не завершены все его основные потоки. В то же время, завершение процесса автоматически прерывает все фоновые потоки[2].

8. Свойства `CurrentCulture` и `CurrentUICulture` имеют тип `CultureInfo` и задают текущую языковую культуру.

Следующий пример демонстрирует настройку свойств потока.

```
var th = new Thread(DoSomeWork)
         {
             Name = "Example Thread",
             Priority = ThreadPriority.BelowNormal,
             IsBackground = true,
             CurrentCulture = new CultureInfo("ru-RU")
         };
```

[1] Любой созданный поток резервирует примерно один мегабайт памяти под свои нужды.

[2] Прекращение работы фонового потока не гарантируется выполнение его блоков `finally`.

Кроме свойств, класс `Thread` содержит методы для управления потоком. Метод `Suspend()` вызывает приостановку потока, метод `Resume()` возобновляет работу потока[1]. Статический метод `Sleep()` приостанавливает выполнение текущего потока на указанное количество миллисекунд или значение `TimeSpan`. Статический метод `Yield()` передаёт управление следующему ожидающему потоку системы. Метод `Join()` позволяет дождаться завершения работы того потока, у которого вызывается. Модификация данного метода блокирует выполнение текущего потока на указанное количество времени.

```
var th = new Thread(DoSomeWork);
th.Start();     // создали и запустили поток
th.Join();      // ждём, пока поток отработает
th.Start();     // запустили поток заново

// если дождались завершения за секунду, res = true
bool res = th.Join(1000);
```

Рис. 14 демонстрирует временную диаграмму работы потоков.

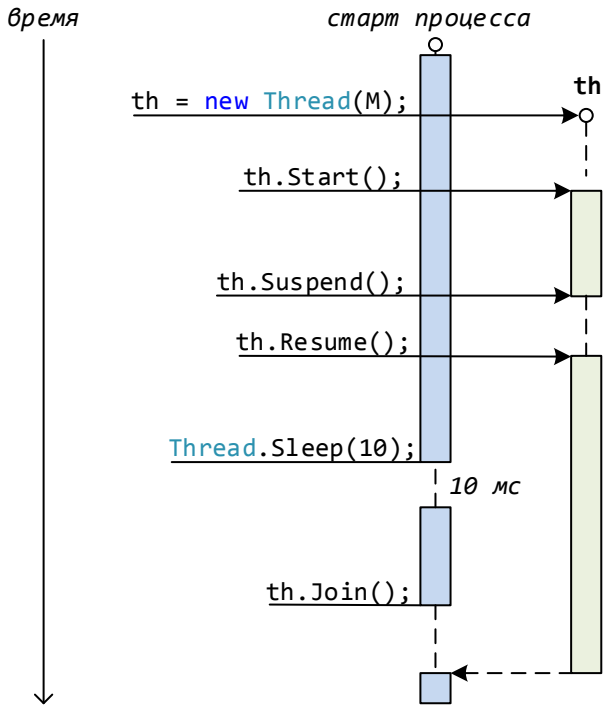

Рис. 14. Временная диаграмма работы потоков.

Для завершения работы выбранного потока используется метод `Abort()`. Данный метод генерирует специальное исключение `ThreadAbortException`. Особенность исключения состоит в том, что его невозможно подавить при помощи `catch`-блока. Исключение может быть отслежено тем потоком, который кто-то

[1] Оба метода – `Suspend()` и `Resume()` – помечены как устаревшие. Использовать их не рекомендуется.

собирается уничтожить, а при помощи статического метода потока `ResetAbort()` запрос на уничтожение можно отклонить.

```
public class MainClass
{
    public static void ThreadProc()
    {
        while (true)
        {
            try
            {
                Console.WriteLine("Do some work...");
                Thread.Sleep(1000);
            }
            catch (ThreadAbortException e)
            {
                // отлавливаем попытку уничтожения и отменяем её
                Console.WriteLine("Somebody tries to kill me!");
                Thread.ResetAbort();
            }
        }
    }

    public static void Main()
    {
        // запускаем поток и ждём три секунды
        var th = new Thread(ThreadProc);
        th.Start();
        Thread.Sleep(3000);

        // пытаемся прервать работу потока th и ждём его завершения
        th.Abort();
        th.Join();

        // ... но не дождёмся, так как поток сам себя "воскресил"
    }
}
```

На рис. 15 показана диаграмма состояний потока с указанием значения свойства `ThreadState`.

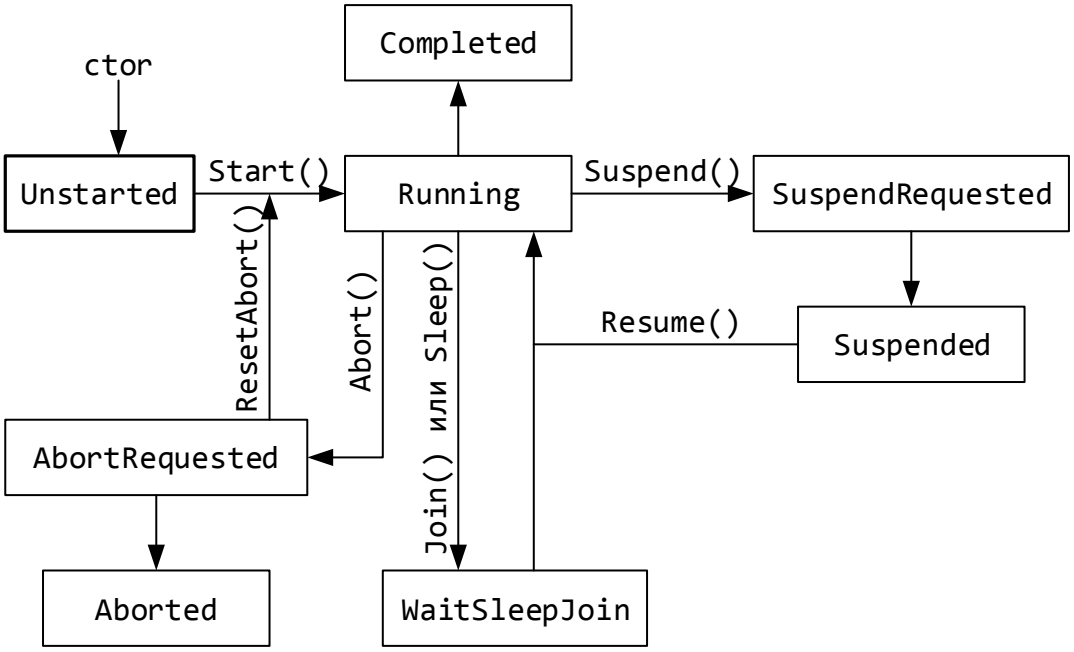

Рис. 15. Диаграмма состояний потока.

Создание отдельного потока – это довольно «затратная» операция с точки зрения расхода времени и памяти. Для уменьшения издержек, связанных с созданием потоков, платформа .NET поддерживает специальный механизм, называемый *пул потоков*. Пул состоит из двух основных элементов: *очереди методов* и *рабочих потоков*. Характеристикой пула является его *ёмкость* – максимальное число рабочих потоков. При работе с пулом метод сначала помещается в очередь. Если у пула есть свободные рабочие потоки, метод извлекается из очереди и направляется свободному потоку для выполнения. Если свободных потоков нет, но ёмкость пула не достигнута, для обслуживания метода формируется новый рабочий поток. Однако этот поток создаётся с задержкой в полсекунды. Если за это время освободится какой-либо из рабочих потоков, то он будет назначен на выполнение метода, а новый рабочий поток создан не будет. Важным нюансом является то, что несколько первых рабочих потоков в пуле создаётся без полусекундной задержки.

Для работы с пулом используется статический класс `ThreadPool`. Метод `SetMaxThreads()` позволяет изменить ёмкость пула. Метод `SetMinThreads()` устанавливает количество рабочих потоков, создаваемых без задержки. Для помещения метода в очередь пула служит метод `QueueUserWorkItem()`. Он принимает делегат типа `WaitCallback`[1] и, возможно, аргумент инкапсулируемого метода.

```
public static void Main()
{
    ThreadPool.QueueUserWorkItem(Go);
    ThreadPool.QueueUserWorkItem(Go, 123);
    Console.ReadLine();
}
```

[1] `public delegate void WaitCallback(object state);`

```
private static void Go(object data)
{
    Console.WriteLine("Hello from the thread pool! " + data);
}
```

Для выполнения в отдельном потоке повторяющегося метода можно применить класс `Timer` из пространства имён `System.Threading`. Конструктор таймера позволяет указать, через какой промежуток времени метод таймера должен выполниться первый раз, а также задать периодичность выполнения (эти величины можно затем изменить при помощи метода `Change()`).

```
using System;
using System.Threading;

public class MainClass
{
    private static bool TickNext = true;

    public static void Main()
    {
        var timer = new Timer(TickTock, null, 1000, 2000);
        Console.WriteLine("Press <Enter> to terminate...");
        Console.ReadLine();
    }

    private static void TickTock(object state)
    {
        Console.WriteLine(TickNext ? "Tick" : "Tock");
        TickNext = !TickNext;
    }
}
```

Заметим, что класс `Timer` работает, используя пул потоков.

32. Синхронизация потоков

При использовании многопоточности естественным образом возникает проблема *синхронизации потоков*, то есть координации их действий для получения предсказуемого результата. В этом параграфе рассматриваются основные средства синхронизации потоков, доступные на платформе .NET.

32.1. Критические секции

Рассмотрим следующий класс:

```
public class ThreadUnsafe
{
    private static int x, y;
```

```
    public void Go()
    {
        if (y != 0)
        {
            Console.WriteLine(x/y);
        }
        y = 0;
    }
}
```

Этот класс небезопасен с точки зрения многопоточного доступа к данным. Если вызвать метод `Go()` в разных потоках одновременно, может возникнуть ошибка деления на ноль, так как поле `y` будет обнулено в одном потоке как раз между проверкой условия `y != 0` и вызовом `Console.WriteLine()` в другом потоке. Чтобы сделать код потокобезопасным, необходимо гарантировать выполнение операторов, составляющих тело метода `Go()`, только одним потоком в любой момент времени. Такие блоки кода называются *критическими секциями*.

Для организации критических секций платформа .NET предлагает статический класс `System.Threading.Monitor`. Метод `Monitor.Enter()` определяет вход в критическую секцию, а метод `Monitor.Exit()` – выход из секции. Вход и выход должны выполняться в одном и том же потоке. Аргументами методов является объект-идентификатор критической секции.

Изменим метод `Go()`, чтобы сделать его потокобезопасным:

```
public class ThreadSafe
{
    // объект locker будет идентификатором критической секции
    private static readonly object locker = new object();
    private static int x, y;

    public void Go()
    {
        Monitor.Enter(locker);   // вход в критическую секцию
        if (y != 0)
        {
            Console.WriteLine(x/y);
        }
        y = 0;
        Monitor.Exit(locker);    // выход из критической секции
    }
}
```

Рис. 16 демонстрирует поведение двух потоков, работающих с одной критической секцией.

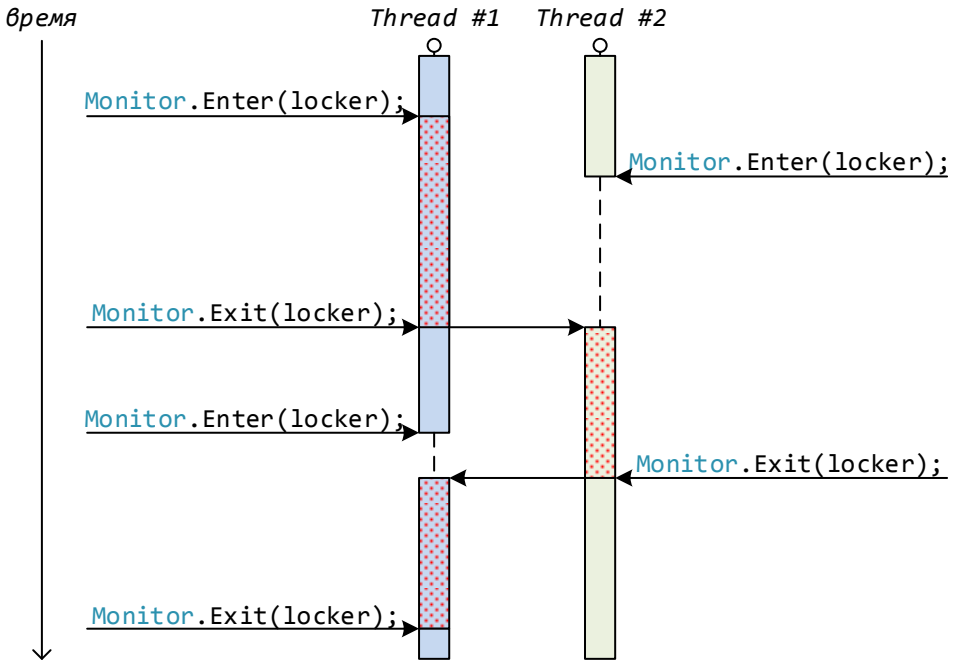

Рис. 16. Два потока работают с одной критической секцией.

Метод `Monitor.Enter()` имеет перегруженную версию, с `ref`-параметром типа `bool`. Если вход в критическую секцию был выполнен успешно, этот параметр возвращается как `true`. Если организовать критическую секцию не удалось (например, по причине недостатка памяти), параметр возвращается как `false`, а метод `Enter()` выбрасывает исключение.

Язык C# содержит оператор `lock`, маскирующий вызовы методов `Monitor.Enter()` и `Monitor.Exit()`. Синтаксис оператора `lock` следующий:

```
lock (выражение) вложенный-оператор
```

Здесь *выражение* должно иметь ссылочный тип и задаёт идентификатор критической секции. Часто в качестве выражения записывают поле или переменную, на которую накладывается блокировка. Оператор `lock` вида `lock (x)` эквивалентен следующему коду:

```
bool lockWasTaken = false;
try
{
    Monitor.Enter(x, ref lockWasTaken);
    // здесь размещается вложенный оператор lock
}
finally
{
    if (lockWasTaken)
    {
        Monitor.Exit(x);
    }
}
```

Перепишем метод `Go()` класса `ThreadSafe`, используя оператор `lock`:

```
public void Go()
{
    lock (locker)
    {
        if (y != 0)
        {
            Console.WriteLine(x/y);
        }
        y = 0;
    }
}
```

Класс `System.Threading.Mutex` (*мьютекс*) подобен классу `Monitor`, но позволяет организовать критическую секцию для нескольких процессов. Применяя `Mutex`, нужно вызвать метод `WaitOne()` для входа в критическую секцию, а метод `ReleaseMutex()` – для выхода из неё (выход может быть произведён только в том же потоке выполнения, что и вход).

Типичный пример использования мьютекса – создание приложения, которое можно запустить только в одном экземпляре:

```
using System;
using System.Threading;

public class OneAtATimePlease
{
    public static void Main()
    {
        // имя мьютекса должно быть уникально для компьютера
        using (var mutex = new Mutex(false, "RandomString"))
        {
            // пытаемся войти в критическую секцию в течение 3 сек.
            // ожидаем 3 секунды на случай, если другой экземпляр
            // приложения в процессе завершения работы
            if (!mutex.WaitOne(TimeSpan.FromSeconds(3), false))
            {
                Console.WriteLine("Another instance is running");
                return;
            }
            RunProgram();
        }
    }

    private static void RunProgram()
    {
        Console.WriteLine("Running (press Enter to exit)");
        Console.ReadLine();
    }
}
```

Семафор – это объект синхронизации, позволяющий войти в заданный участок кода не более чем N потокам (N – *ёмкость семафора*). Аналогией семафора является охранник у входа в клуб с фиксированным количеством мест. Новые посетители попадают в заполненный клуб, только если из него кто-то ушёл. Семафор с ёмкостью, равной единице, аналогичен монитору или мьютексу, однако получение и снятие блокировки в случае семафора может выполняться из разных потоков.

Для организации семафоров платформа .NET предлагает классы `Semaphore` и `SemaphoreSlim` из пространства имён `System.Threading`. Первый класс применяется для синхронизации между процессами, второй работает только в рамках одного процесса. Метод `Wait()` этих классов выполняет получение блокировки, а метод `Release()` – снятие блокировки.

```
using System;
using System.Threading;

public class TheClub
{
    // ёмкость семафора равна 2
    private static SemaphoreSlim s = new SemaphoreSlim(2);

    public static void Main()
    {
        for (var i = 1; i <= 4; i++)
        {
            new Thread(Enter).Start(i);
        }
    }

    private static void Enter(object id)
    {
        Console.WriteLine(id + " wants to enter");
        s.Wait();

        Console.WriteLine(id + " is in!");     // только два потока
        Thread.Sleep(1000 * (int) id);         // могут одновременно
        Console.WriteLine(id + " is leaving"); // выполнять этот код

        s.Release();
    }
}
```

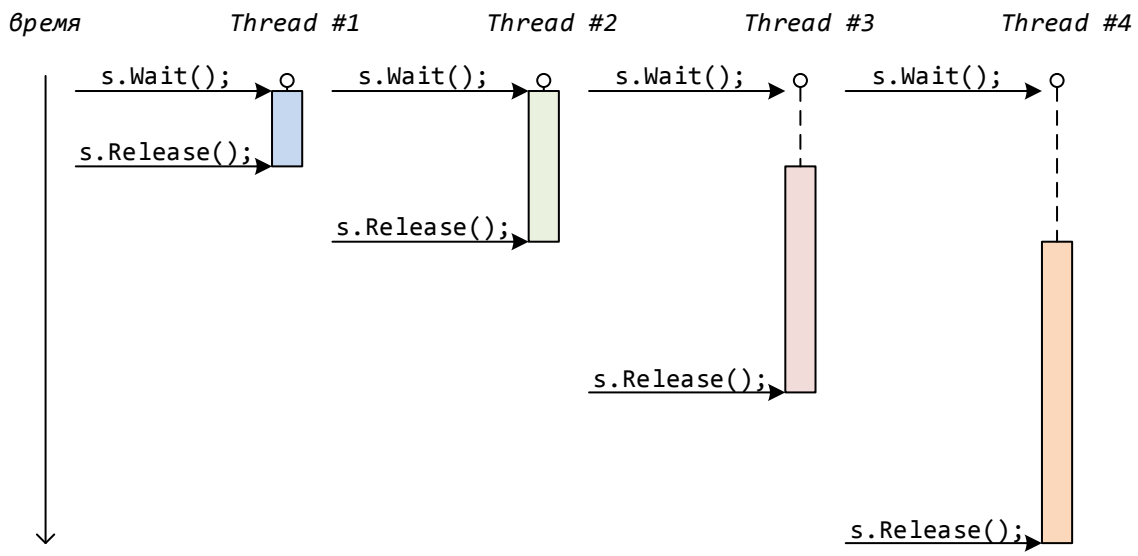

Рис. 17. Демонстрация работы семафора в классе `TheClub`.

Иногда ресурс нужно блокировать так, чтобы читать его могли несколько потоков, а записывать – только один. Для этих целей предназначен класс `ReaderWriterLockSlim`. Его экземплярные методы `EnterReadLock()` и `ExitReadLock()` задают секцию чтения ресурса, а методы `EnterWriteLock()` и `ExitWriteLock()` – секцию записи ресурса. В следующем примере при помощи `ReaderWriterLockSlim` построен класс, реализующий простой кэш с поддержкой многопоточности:

```
public class SynchronizedCache
{
    private Dictionary<int, string> cache =
                                    new Dictionary<int, string>();
    private ReaderWriterLockSlim locker =
                                    new ReaderWriterLockSlim();

    public string Read(int key)
    {
        locker.EnterReadLock();     // секция чтения началась
        try
        {
            return cache[key];      // читать могут несколько потоков
        }
        finally
        {
            locker.ExitReadLock(); // секция чтения закончилась
        }
    }

    public void Add(int key, string value)
    {
        locker.EnterWriteLock();        // секция записи началась
```

```
            try
            {
                cache.Add(key, value);
            }
            finally
            {
                locker.ExitWriteLock();    // секция запись закончилась
            }
        }

        public bool AddWithTimeout(int key, string value, int timeout)
        {
            if (locker.TryEnterWriteLock(timeout))    // таймаут входа
            {
                try
                {
                    cache.Add(key, value);
                }
                finally
                {
                    locker.ExitWriteLock();
                }
                return true;
            }
            return false;
        }
    }
```

32.2. Синхронизация на основе подачи сигналов

При использовании *синхронизации на основе подачи сигналов* один поток получает уведомления от другого потока. Обычно уведомления используются для возобновления работы заблокированного потока. Для реализации данного подхода платформа .NET предлагает классы `AutoResetEvent`, `ManualResetEvent`, `ManualResetEventSlim`, `CountdownEvent` и `Barrier` (все они размещены в пространстве имён `System.Threading`).

Классы `AutoResetEvent`, `ManualResetEvent`, `ManualResetEventSlim` унаследованы от класса `EventWaitHandle`. Имея доступ к объекту `EventWaitHandle`, поток может вызвать его метод `WaitOne()`, чтобы остановиться и ждать сигнала. Для отправки сигнала применяется вызов метода `Set()`. Если используются `ManualResetEvent` и `ManualResetEventSlim`, все ожидающие потоки освобождаются и продолжают выполнение. При использовании `AutoResetEvent` ожидающие потоки освобождаются и запускаются последовательно, на манер очереди. Аналогией для `AutoResetEvent` является турникет, пропускающий людей по одному, а `ManualResetEvent` подобен воротам, которые или закрыты, или открыты.

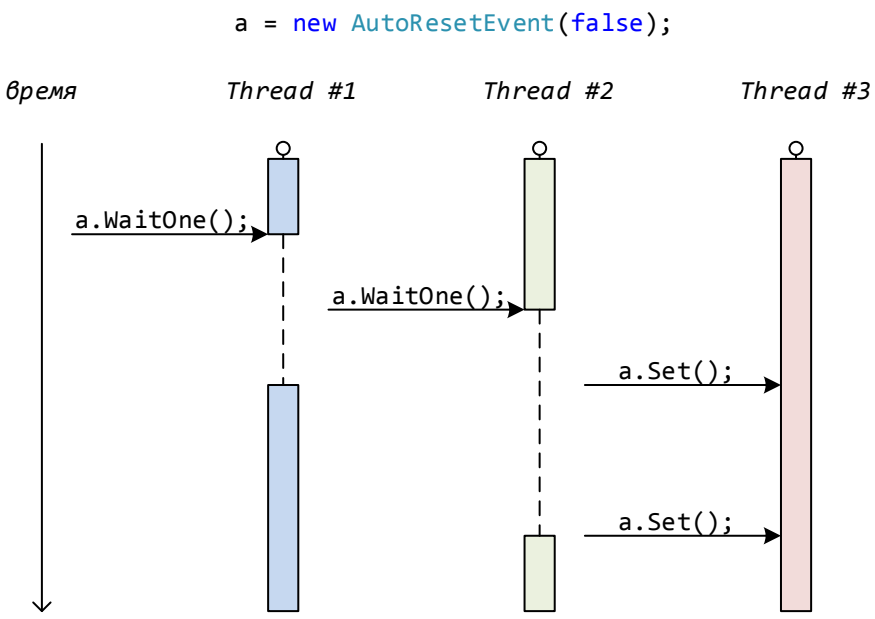

Рис. 18. Пример работы с объектом `AutoResetEvent`.

Класс `CountdownEvent` позволяет организовать счётчик уведомлений. Конструктор класса принимает в качестве аргумента начальное значение счётчика. Вызов экземплярного метода `Signal()` уменьшает значение счётчика на единицу. Метод `Wait()` блокирует поток, пока счётчик не станет равным нулю.

```
private static CountdownEvent counter = new CountdownEvent(3);

public static void Main()
{
    new Thread(SaySomething).Start("I am thread 1");
    new Thread(SaySomething).Start("I am thread 2");
    new Thread(SaySomething).Start("I am thread 3");

    // ждём, пока метод Signal() не вызовется три раза
    counter.Wait();
    Console.WriteLine("All threads have finished speaking!");
}

private static void SaySomething(object thing)
{
    Thread.Sleep(1000);
    Console.WriteLine(thing);
    counter.Signal();
}
```

Объект класса `Barrier` организует для нескольких потоков точку встречи во времени. Чтобы использовать `Barrier`, нужно вызвать его конструктор, передав количество встречающихся потоков. Затем отдельные потоки вызывают экземплярный метод `Barrier.SignalAndWait()`, чтобы приостановиться и продолжить выполнение после совместной встречи.

В следующем примере каждый из трёх потоков печатает числа от 0 до 4, синхронизируя работу со своими «коллегами»:

```
using System;
using System.Threading;

public class BarrierExample
{
    private static readonly Barrier _barrier = new Barrier(3);

    public static void Main()
    {
        new Thread(Speak).Start();
        new Thread(Speak).Start();
        new Thread(Speak).Start();
        // вывод: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4

        Console.ReadLine();
    }

    private static void Speak()
    {
        for (var i = 0; i < 5; i++)
        {
            Console.Write(i + " ");
            _barrier.SignalAndWait();
        }
    }
}
```

32.3. Неблокирующие средства синхронизации

Неблокирующие средства синхронизации позволяют осуществлять совместный доступ к простым ресурсам нескольких потоков без блокировки, паузы или ожидания. Коротко рассмотрим неблокирующие средства синхронизации, которые предлагает платформа .NET.

Для оптимизации выполнения программы центральный процессор иногда применяет перестановку инструкций программы или их кэширование. Чтобы отменить подобную оптимизацию следует разместить в исходном коде *барьеры памяти*. Процессор не способен изменить порядок команд, чтобы инструкции до барьера, выполнялись после инструкций за барьером. Для установки барьера памяти нужно вызвать статический метод `Thread.MemoryBarrier()`. Язык C# содержит специальный модификатор `volatile`, применяемый при объявлении поля. Инструкции записи данных в такое не переставляются процессором в целях оптимизации.

Статический класс `System.Threading.Interlocked` имеет методы для инкремента, декремента и сложения аргументов типа `int` или `long`, а также методы

присваивания значений числовым и ссылочным переменным. Каждый метод гарантировано выполняется как *атомарная операция*, без блокировки текущего потока выполнения.

```
int x = 10, y = 20;
Interlocked.Add(ref x, y);                    // x = x + y
Interlocked.Increment(ref x);                 // x++
Interlocked.Exchange(ref x, y);               // x = y
Interlocked.CompareExchange(ref x, 50, y);    // if (x == y) x = 50
```

32.4. Разделение данных между потоками

Если некий метод запускается в нескольких потоках, только локальные переменные метода будут уникальными для потока. Поля объектов по умолчанию разделяются между всеми потоками. В пространстве имён `System` определён атрибут `[ThreadStatic]`, применяемый к статическим полям. Если поле помечено таким атрибутом, то каждый поток будет содержать свой экземпляр поля. Для `[ThreadStatic]`-полей не рекомендуется делать инициализацию при объявлении, так как код инициализации выполнится только в одном потоке.

```
public class SomeClass
{
        public static int SharedField = 25;

        [ThreadStatic]
        public static int NonSharedField;
}
```

Для создания неразделяемых статических полей можно использовать тип `ThreadLocal<T>`. Перегруженный конструктор `ThreadLocal<T>` принимает функцию инициализации поля. Значение поля хранится в свойстве `Value`.

```
public class Slot
{
    private static readonly Random rnd = new Random();

    private static int Shared = 25;
    private static ThreadLocal<int> NonShared =
                   new ThreadLocal<int>(() => rnd.Next(1, 20));

    public static void PrintData()
    {
        Console.WriteLine("Thread: {0} Shared: {1} NonShared: {2}",
                          Thread.CurrentThread.Name,
                          Shared, NonShared.Value);
    }
}
```

```
public class MainClass
{
    public static void Main()
    {
        // для тестирования запускаем три потока
        new Thread(Slot.PrintData) {Name = "First"}.Start();
        new Thread(Slot.PrintData) {Name = "Second"}.Start();
        new Thread(Slot.PrintData) {Name = "Third"}.Start();

        Console.ReadLine();
    }
}
```

Отметим, что класс `Thread` имеет статические методы `AllocateDataSlot()`, `AllocateNamedDataSlot()`, `GetNamedDataSlot()`, `FreeNamedDataSlot()`, `GetData()`, `SetData()`, которые предназначены для работы с *локальными хранилищами данных* потока. Эти локальные хранилища могут рассматриваться как альтернатива неразделяемым статическим полям.

Распространённый шаблон при разработке многопоточных приложений – *неизменяемый объект* (immutable object). После создания такой объект допускает только чтение своих полей, но не запись. Приведём пример класса, объекты которого являются неизменяемыми:

```
public class ProgressStatus
{
    public readonly int PercentComplete;
    public readonly string StatusMessage;

    public ProgressStatus(int percentComplete, string statusMessage)
    {
        PercentComplete = percentComplete;
        StatusMessage = statusMessage;
    }
}
```

Достоинство неизменяемых объектов с точки зрения многопоточности заключается в том, что работа с ними требует коротких блокировок, обычно обрамляющих операции присваивания объектов:

```
public class WorkWithImmutable
{
    private readonly object _locker = new object();
    private ProgressStatus _status;

    public void SetFields()
    {
        // создаём и настраиваем временный объект
        var status = new ProgressStatus(50, "Working on it");
```

```
            // переносим информацию, используя короткую блокировку
            lock (_locker) _status = status;
        }

        public void ReadInfo()
        {
            // используя короткую блокировку, создаём временную копию
            ProgressStatus statusCopy;
            lock (_locker) statusCopy = _status;

            // работаем с копией
            int pc = statusCopy.PercentComplete;
            string msg = statusCopy.StatusMessage;
        }
    }
```

33. Выполнение асинхронных операций при помощи задач

Поток, рассматриваемый как объект класса `Thread`, – это низкоуровневый инструмент для организации параллельной работы, и, будучи таковым, он обладает рядом ограничений. В частности, у потоков отсутствует механизм *продолжений*, когда после завершения метода, работающего в потоке, в этом же потоке автоматически запускается другой заданный метод. Затруднено получение значения функции, выполняющейся в отдельном потоке. Наконец, необдуманное создание множества потоков ведёт к повышенному расходу памяти и замедлению работы приложения. *Задача* (task) – это сущность, в целом подобная потоку. Однако, являясь абстракцией более высокого уровня, она призвана устранить указанные выше ограничения потоков. Концепция задач была представлена в четвёртой версии платформы .NET.

33.1. Базовые сведения о задачах

Для описания задач используются объекты классов `Task` и `Task<TResult>`, размещённых в пространстве имён `System.Threading.Tasks`. Табл. 20 содержит информацию об элементах класса `Task`.

Таблица 20

Элементы класса `Task`

Имя элемента	Описание
`AsyncState`	Объект, заданный при создании задачи (аргумент `Action<object>`)
`ConfigureAwait()`	Настраивает объект ожидания, используемый для текущей задачи
`ContinueWith()`	Используются для указания метода, выполняемого после завершения текущей задачи
`CreationOptions`	Опции, указанные при создании задачи (тип `TaskCreationOptions`)
`CurrentId`	Статическое свойство типа `int?`, которое возвращает целочисленный идентификатор текущей задачи
`Delay()`	Статический метод; позволяет создать задачу с указанной задержкой старта
`Dispose()`	Освобождение ресурсов, связанных с задачей

Окончание табл. 20

`Exception`	Возвращает объект типа `AggregateException`, который соответствует исключению, прервавшему выполнение задачи
`Factory`	Доступ к фабрике, содержащей методы создания `Task` и `Task<T>`
`GetAwaiter()`	Получает объект ожидания для текущей задачи
`Id`	Целочисленный идентификатор задачи
`IsCanceled`	Булево свойство, указывающее, была ли задача отменена
`IsCompleted`	Свойство равно `true`, если задача успешно завершилась
`IsFaulted`	Свойство равно `true`, если задача сгенерировала исключение
`Run()`	Статический метод; выполняет создание и запуск задачи
`RunSynchronously()`	Запуск задачи синхронно
`Start()`	Запуск задачи асинхронно
`Status`	Возвращает текущий статус задачи (объект типа `TaskStatus`)
`Wait()`	Приостанавливает текущий поток до завершения задачи
`WaitAll()`	Статический метод; приостанавливает текущий поток до завершения всех указанных задач
`WaitAny()`	Статический метод; приостанавливает текущий поток до завершения любой из указанных задач
`WhenAll()`	Статический метод; создаёт задачу, которая будет выполнена после выполнения всех указанных задач
`WhenAny()`	Статический метод; создаёт задачу, которая будет выполнена после выполнения любой из указанных задач

Простейший способ создания и запуска задачи – вызов статического метода `Task.Run()`. Этот метод принимает в качестве аргумента объект типа `Action`.

```
var task = Task.Run(() => Console.WriteLine("Hello"));
```

Класс `TaskFactory` содержит набор методов, соответствующих некоторым сценариям использования задач – `StartNew()`, `FromAsync()`, `ContinueWhenAll()`, `ContinueWhenAny()`. Экземпляр `TaskFactory` доступен через статическое свойство `Task.Factory`. Вызов `Task.Run()` – это сокращение для `Task.Factory.StartNew()`.

```
// аналог предыдущего примера
Action action = () => Console.WriteLine("Hello");
var task = Task.Factory.StartNew(action);
```

Для создания задачи можно использовать один из перегруженных конструкторов класса `Task`. При этом указывается аргумент типа `Action` – метод, выполняемый в задаче. Если необходимо выполнить метод с параметром, используется `Action<object>` и дополнительный аргумент типа `object`.

```
Action action = () => Console.WriteLine("Hello");
Action<object> method = x => {
                                Thread.Sleep(1000);
                                Console.WriteLine(x.ToString());
                            };

var task1 = new Task(action);
var task2 = new Task(method, 25);
```

Перегруженные конструкторы класса `Task` принимают опциональные аргументы типа `CancellationToken` и `TaskCreationOptions`. Перечисление `TaskCreationOptions` задаёт вид задачи (например, `LongRunning` – долгая задача). Структура `CancellationToken` применяется для прерывания задачи (это так называемый *токен отмены*).

```
var task1 = new Task(action, TaskCreationOptions.LongRunning);
```

Метод `Start()` запускает задачу, вернее, помещает её в очередь запуска *планировщика задач*. По умолчанию применяется планировщик на основе пула потоков, но существует возможность использовать пользовательский планировщик[1]. Метод `RunSynchronously()` выполняет задачу синхронно (в терминах используемого планировщика задач).

```
task1.Start();                  // асинхронный запуск
task2.RunSynchronously();       // синхронный запуск
```

Методы класса `Task` `Wait()`, `WaitAll()` и `WaitAny()` останавливают основной поток до завершения задачи (или задач). Перегруженные версии методов позволяют указать период ожидания завершения и токен отмены.

```
task1.Wait(1000);
Task.WaitAll(task1, task2);
```

Класс `Task<TResult>` наследуется от `Task` и описывает задачу, возвращающую значение типа `TResult`. Дополнительно к элементам базового класса, `Task<TResult>` объявляет свойство `Result` для хранения вычисленного значения. Конструкторы класса `Task<TResult>` принимают аргументы типа `Func<TResult>` и `Func<object, TResult>` (опционально – аргументы типа `CancellationToken` и `TaskCreationOptions`).

```
Func<int> func = () => {
                            Thread.Sleep(2000);
                            return 100;
                        };

var task = new Task<int>(func);
Console.WriteLine(task.Status);         // Created

task.Start();
Console.WriteLine(task.Status);         // WaitingToRun

task.Wait();
Console.WriteLine(task.Result);         // 100
```

[1] По адресу code.msdn.microsoft.com/ParExtSamples доступен для загрузки код, содержащий примеры нестандартных планировщиков задач (см. ссылку на Parallel Extensions Extras).

33.2. Обработка исключений и отмена выполнения задач

В отличие от потоков, задачи легко распространяют исключения. Если код внутри задачи генерирует необработанное исключение (задача *отказывает*), это исключение автоматически повторно сгенерируется при вызове метода `Wait()` или при доступе к свойству `Result` класса `Task<TResult>`. CLR помещает исключение в оболочку `System.AggregateException`. При параллельном возникновении нескольких исключений все они собираются в единое исключение `AggregateException` и доступны в свойстве-коллекции `InnerExceptions`.

```
var task = Task.Run(() => { throw new Exception(); });
try
{
    task.Wait();
}
catch (AggregateException ex)
{
    var message = ex.InnerException.Message;
    Console.WriteLine(message);
}
```

При работе с задачами применяется особый подход для отмены их выполнения. Структура типа `CancellationToken` (токен отмены) – это своеобразный маркер того, что задачу можно отменить. Токен отмены принимает в качестве аргумента, в частности, конструктор задачи. Класс `CancellationTokenSource` содержит свойство `Token` для получения токенов отмены и метод `Cancel()` для отмены выполнения всех задач, использующих общий токен.

В следующем фрагменте кода демонстрируется типичный сценарий использования токенов отмены: вначале создаётся объект `CancellationTokenSource`, затем его токен назначается нескольким задачам, а потом вызывается метод `Cancel()`, прерывающий выполнение всех этих задач:

```
var tokenSource = new CancellationTokenSource();

// используем один токен в двух задачах
new Task(SomeMethod, tokenSource.Token).Start();
new Task(OtherMethod, tokenSource.Token).Start();

// в нужный момент отменяем обе задачи
tokenSource.Cancel();
```

33.3. Продолжения

Продолжение сообщает задаче, что после её завершения она должна продолжить делать что-то другое. Первый способ организации продолжения заключается в использовании экземплярного метода задачи `ContinueWith()`.

```
var task1 = Task.Run(() => Console.Write("Task.."));
var task2 = task1.ContinueWith(t => Console.Write("continuation"));
```

После того как задача `task1` (*предшественник*) завершается, отказывает или отменяется, задача `task2` (*продолжение*) запускается. Аргумент `t`, переданный лямбда-выражению продолжения – это ссылка на предшествующую задачу[1].

Выполнение продолжения можно запланировать на основе завершения множества предшествующих задач при помощи статических методов `Task.WhenAll()` и `Task.WhenAny()`.

```
var task1 = Task.Run(() => Console.Write("X"));
var task2 = Task.Run(() => Console.Write("Y"));
var continuation = Task.WhenAll(task1, task2)
                       .ContinueWith(t => Console.Write("Done"));
```

Второй способ организации продолжения заключается в использовании объекта ожидания. *Объект ожидания* – это любой объект, имеющий методы `OnCompleted()` и `GetResult()` и булево свойство `IsCompleted`. Вызов метода `GetAwaiter()` на задаче возвращает объект ожидания. Метод `OnCompleted()` принимает в качестве аргумента делегат, содержащий код продолжения. Метод `GetResult()` возвращает результат работы предшественника (или `void`).

```
// задача подсчёта простых чисел (используется LINQ)
Task<int> prime = Task.Run(() =>
    Enumerable.Range(2, 3000000 - 2).Count(n =>
    Enumerable.Range(2, (int) Math.Sqrt(n) - 1).All(i => n%i > 0)));

// получаем объект продолжения
var awaiter = prime.GetAwaiter();

// указываем, что делать после окончания предшественника
awaiter.OnCompleted(() =>
        {
            // получаем результат вычислений предшественника
            int result = awaiter.GetResult();
            Console.WriteLine(result);
        });
```

33.4. Асинхронные функции в C#

Для поддержки асинхронного программирования в версии C# 5.0 появилась операция `await`. Синтаксис этой операции следующий:

```
var результат = await выражение;
оператор(ы);
```

Компилятор разворачивает приведённую выше конструкцию в такой функциональный эквивалент:

---

[1] По умолчанию предшественник и продолжение могут выполняться в разных потоках. Чтобы они выполнялись в одном потоке, передайте методу `ContinueWith()` дополнительный аргумент со значением `TaskContinuationOptions.ExecuteSynchronously`.

```
var awaiter = выражение.GetAwaiter();
awaiter.OnCompleted(()=>
        {
            var результат = awaiter.GetResult();
            оператор(ы);
}
```

Операция `await` применима и к выражению, не возвращающему значения:

```
await выражение;
оператор(ы);
```

Выражение, на котором применяется `await`, обычно является задачей. Тем не менее, компилятор удовлетворит любой объект ожидания (определение данного понятия приведено выше). Операция `await` может применяться только внутри метода (или лямбда-выражения) со специальным модификатором `async`. Метод должен возвращать `void` либо тип `Task` или `Task<TResult>`.

Методы с модификатором `async` называются *асинхронными функциями*. Встретив выражение `await`, процесс выполнения (обычно) производит возврат в вызывающий код – почти как оператор `yield return` в итераторе. Но перед возвратом исполняющая среда присоединяет к ожидающей задаче признак продолжения, гарантирующий, что когда задача завершается, управление перейдёт обратно в метод и продолжится с места, в котором оно его оставило.

Рассмотрим несколько примеров использования операции `await`. Пусть имеется обычный метод, выполняющий чтение содержимого сайта. Для этого используется класс `System.Net.WebClient` и его метод `DownloadString()`.

```
private void ReadFromWeb()
{
    var web = new WebClient();
    var text = web.DownloadString("http://msdn.com");
    Console.WriteLine(text.Length);
}
```

Вызов метода `ReadFromWeb()` задержит основной поток выполнения на несколько миллисекунд (или даже секунд – зависит от скорости сети). К счастью, создатели класса `WebClient` включили в него метод `DownloadStringTaskAsync()`. Этот метод создаёт *задачу* `Task<string>` для чтения сайта и возвращает управление. Данный факт позволяет нам превратить метод `ReadFromWeb()` в асинхронную функцию, которая не блокирует при вызове основной поток.

```
private async void ReadFromWeb()
{
    var web = new WebClient();
    var text = await web.DownloadStringTaskAsync("http://msdn.com");
    Console.WriteLine(text.Length);
}
```

В стандартных классах платформы .NET многие методы, выполняющие долгие операции, получили поддержку в виде асинхронных аналогов. Обычно эти новые методы можно распознать по суффиксу *Async* в названии (например, метод для чтения `ReadLineAsync()` из класса `StreamReader`). Этот факт позволяет перенести идеи предыдущего примера на другие подобные ситуации.

Следующий пример показывает, что модификатор `async` применим не только к именованным методам, но и к лямбда-выражениям:

```
Func<Task> unnamed = async () =>
            {
                // Task.Delay() - асинхронный аналог Thread.Sleep()
                await Task.Delay(1000);
                Console.Write("Done");
            };
```

34. Платформа параллельных вычислений

Платформа параллельных вычислений (Parallel Framework, PFX) – это набор типов и технологий, являющийся частью платформы .NET. PFX предназначена для повышения производительности разработчиков за счёт средств, упрощающих добавление параллелизма в приложения. PFX динамически масштабирует степень параллелизма для наиболее эффективного использования всех доступных процессорных ядер.

PFX обеспечивает три уровня организации параллелизма:

1. *Параллелизм на уровне задач*. Частью PFX является *библиотека параллельных задач* (Task Parallel Library, TPL). Эта библиотека представляет собой набор типов в пространствах имён `System.Threading.Tasks` и `System.Threading`. TPL обеспечивает высокоуровневую работу с пулом потоков, позволяя явно структурировать параллельно исполняющийся код с помощью легковесных задач. Планировщик библиотеки выполняет диспетчеризацию задач, а также предоставляет единообразный механизм отмены задач и обработки исключительных ситуаций. Работа с задачами рассмотрена в предыдущем параграфе.

2. *Параллелизм при императивной обработке данных*. PFX содержит параллельные реализации основных итеративных операторов, таких как циклы `for` и `foreach`. При этом выполнение автоматически распределяется на все процессорные ядра вычислительной системы.

3. *Параллелизм при декларативной обработке данных* реализуется при помощи параллельного интегрированного языка запросов (PLINQ). PLINQ выполняет запросы LINQ параллельно, обеспечивая масштабируемость и загрузку процессорных ядер.

34.1. Параллелизм при императивной обработке данных

Класс `System.Threading.Tasks.Parallel` позволяет распараллеливать циклы и последовательность блоков кода. Эта функциональность реализована как набор статических методов `For()`, `ForEach()` и `Invoke()`.

Методы `Parallel.For()` и `Parallel.ForEach()` являются параллельными аналогами циклов `for` и `foreach`. Их использование корректно в случае независимости итераций цикла, то есть, если ни в одной итерации не используется результаты работы предыдущих итераций.

Существует несколько перегруженных вариантов метода `Parallel.For()`, однако любой из них подразумевает указание начального и конечного значения счётчика (типа `int` или `long`) и тела цикла в виде объекта делегата. В качестве примера использования `Parallel.For()` приведём код, перемножающий две квадратные матрицы (`m1` и `m2` – исходные матрицы, `result` – их произведение, `size` – размер матриц):

```
Parallel.For(0, size, i =>
        {
            for (int j = 0; j < size; j++)
            {
                result[i, j] = 0;
                for (int k = 0; k < size; k++)
                {
                    result[i, j] += m1[i, k] * m2[k, j];
                }
            }
        });
```

Метод `Parallel.ForEach()` имеет множество перегрузок. Простейший вариант предполагает указание коллекции, реализующей `IEnumerable<T>`, и объекта `Action<T>`, описывающего тело цикла:

```
Parallel.ForEach(Directory.GetFiles(path, "*.jpg"),
                 image => Process(image));
```

Статический метод `Parallel.Invoke()` позволяет распараллелить исполнение блоков операторов. Часто в приложениях существуют такие последовательности операторов, для которых не имеет значения порядок выполнения операторов внутри них. В таких случаях вместо последовательного выполнения операторов друг за другом допустимо их параллельное выполнение, позволяющее сократить время решения задачи. В базовом варианте метод `Invoke()` имеет параметр-список объектов `Action`:

```
// используем Invoke() для параллельной загрузки файлов
Parallel.Invoke(
    () => new WebClient().DownloadFile("http://www.linqpad.net",
                                       "lp.html"),
    () => new WebClient().DownloadFile("http://www.jaoo.dk",
                                       "jaoo.html"));
```

Заметим, что каждый из методов `For()`, `ForEach()` и `Invoke()` может принимать аргумент типа `ParallelOptions` для настройки поведения метода.

34.2. Параллелизм при декларативной обработке данных

PLINQ (Parallel Language Integrated Query) – реализация LINQ, в которой запросы выполняются параллельно. PLINQ поддерживает большинство операторов LINQ to Objects и имеет минимальное влияние на имеющуюся модель LINQ.

Начнём рассмотрение PLINQ со следующего примера. Найдём все простые числа от 3 до 100000, используя простой алгоритм с операторами LINQ:

```
var numbers = Enumerable.Range(3, 100000 - 3);
var parallelQuery =
    from n in numbers.AsParallel()
    where Enumerable.Range(2, (int) Math.Sqrt(n)).All(i => n%i > 0)
    select n;
var primes = parallelQuery.ToArray();
```

В приведённом фрагменте кода использовался метод `AsParallel()`. Это метод расширения из статического класса `System.Linq.ParallelEnumerable`. Метод `AsParallel()` конвертирует коллекцию `IEnumerable<T>` в коллекцию `ParallelQuery<T>`. Класс `ParallelEnumerable` имеет набор методов расширения для `ParallelQuery<T>`, соответствующих операторам LINQ, но использующих для выполнения механизм задач.

Кроме `AsParallel()`, класс `ParallelEnumerable` содержит ещё несколько особых методов:

1. `AsSequential()` – конвертирует объект `ParallelQuery<T>` в коллекцию `IEnumerable<T>` так, что все запросы выполняются последовательно.
2. `AsOrdered()` – при параллельной обработке заставляет сохранять в `ParallelQuery<T>` порядок элементов (это замедляет обработку).
3. `AsUnordered()` – при параллельной обработке позволяет игнорировать в `ParallelQuery<T>` порядок элементов (отмена вызова `AsOrdered()`).
4. `WithCancellation()` – устанавливает для `ParallelQuery<T>` указанное значение токена отмены.
5. `WithDegreeOfParallelism()` – указывает для `ParallelQuery<T>`, на сколько параллельных частей нужно разбивать коллекцию для обработки.
6. `WithExecutionMode()` – задаёт опции выполнения параллельных запросов в виде перечисления `ParallelExecutionMode`.

Приведём ещё один пример использования PLINQ. Будем вычислять времени отклика от шести заданных сайтов. При этом явно установим степень параллелизма. В примере используется класс `Ping` из пространства имён `System.Net.NetworkInformation`.

```
var siteNames = new[] {"www.tut.by", "habrahabr.ru",
                       "www.takeonit.com", "stackoverflow.com"};

var pings = from site in siteNames.AsParallel()
                                  .WithDegreeOfParallelism(6)
            let p = new Ping().Send(site)
            select new {site, Time = p.RoundtripTime};
```

```
foreach (var ping in pings)
{
    Console.WriteLine("Site {0} has ping {1}",
                      ping.site, ping.Time);
}
```

34.3. Коллекции, поддерживающие параллелизм

Библиотека параллельных расширений содержит набор классов, представляющих коллекции с различным уровнем поддержки параллелизма. Указанные классы сосредоточены в пространстве имён `System.Collections.Concurrent`.

Класс `BlockingCollection<T>` является реализацией шаблона «поставщик-потребитель». Этот класс реализует интерфейсы `IEnumerable<T>`, `ICollection`, `IDisposable` и имеет собственные элементы, описанные в табл. 21.

Таблица 21

Элементы класса `BlockingCollection<T>`

Имя элемента	Описание
`Add()`	Добавляет элемент в коллекцию
`AddToAny()`	Статический метод, который добавляет элемент в любую из указанных `BlockingCollection<T>`
`CompleteAdding()`	После вызова этого метода добавление элементов невозможно
`GetConsumingEnumerable()`	Возвращает перечислитель, который перебирает элементы с их одновременным удалением из коллекции
`Take()`	Получает элемент и удаляет его из коллекции. Если коллекция пуста, и у коллекции был вызван метод `CompleteAdding()`, генерируется исключение
`TakeFromAny()`	Статический метод, который получает элемент из любой указанной `BlockingCollection<T>`
`TryAdd()`	Пытается добавить элемент в коллекцию, в случае успеха возвращает `true`. Дополнительно может быть задан временной интервал и токен отмены
`TryAddToAny()`	Статический метод, который пытается добавить элемент в любую из указанных коллекций
`TryTake()`	Пытается получить элемент (с удалением из коллекции), в случае успеха возвращает `true`
`TryTakeFromAny()`	Статический метод, который пытается получить элемент из любой указанной `BlockingCollection<T>`
`BoundedCapacity`	Свойство возвращает максимальное число элементов, которое можно добавить в коллекцию без блокировки поставщика (данный параметр может быть задан при вызове конструктора `BlockingCollection<T>`)
`IsAddingCompleted`	Возвращает `true`, если вызывался `CompleteAdding()`
`IsCompleted`	Возвращает `true`, если вызывался `CompleteAdding()`, и коллекция пуста

Продемонстрируем работу с `BlockingCollection<T>`, используя десять задач в качестве поставщика и одну в качестве потребителя:

```
BlockingCollection<int> bc = new BlockingCollection<int>();
for (int producer = 0; producer < 10; producer++)
{
    Task.Factory.StartNew(() =>
    {
        Random rand = new Random();
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(200);
            bc.Add(rand.Next(100));
        }
    });
}

var consumer = Task.Factory.StartNew(() =>
                {
                    foreach (var item in bc.GetConsumingEnumerable())
                    {
                        Console.WriteLine(item);
                    }
                });
consumer.Wait();
```

Классы `ConcurrentQueue<T>`, `ConcurrentStack<T>`, `ConcurrentBag<T>` и `ConcurrentDictionary<T>` – это потокобезопасные классы для представления очереди, стека, неупорядоченного набора объектов и словаря. Предполагается, что данные классы будут использоваться в качестве ресурсов, разделяемых между потоками, вместо обычных классов-коллекций. Отличительная особенность данных коллекций – наличие `Try`-методов для получения (изменения) элементов. Такие методы удобны, так как исключают предварительные проверки существования и необходимость использования в клиентском коде секции `lock`.

```
// используем объект типа ConcurrentStack<T>
T item;
if (concurrentStack.TryPop(out item))    // пытаемся извлечь элемент
{
    UseData(item);
}
```

35. Асинхронный вызов методов

Платформа .NET содержит средства для поддержки асинхронного вызова методов. При *асинхронном вызове* поток выполнения разделяется на две части: в одной выполняется метод, а в другой – процесс программы. Асинхронный вызов служит альтернативой использованию многопоточности явным образом[1].

[1] Подход, описанный в данном параграфе, является старой моделью организации параллельных вычислений. Вместо работы с асинхронными методами следует применять выполнение асинхронных операций при помощи задач.

Асинхронный вызов всегда выполняется посредством объекта некоторого делегата. Любой такой объект содержит два специальных метода для асинхронных вызовов – `BeginInvoke()` и `EndInvoke()`. Эти методы генерируются во время выполнения программы, так как их сигнатура зависит от делегата. Внимание: объект группового делегата нельзя вызвать асинхронно.

Метод `BeginInvoke()` обеспечивает асинхронный запуск. Кроме параметров, указанных при описании делегата, метод `BeginInvoke()` имеет два дополнительных параметра. Первый дополнительный параметр указывает на *функцию завершения*, выполняемую после окончания асинхронного метода. Второй дополнительный параметр – это объект, при помощи которого функции завершения может быть передана некоторая информация. Метод `BeginInvoke()` возвращает объект, реализующий интерфейс `IAsyncResult`. При помощи этого объекта становится возможным различать асинхронные вызовы одного и того же метода.

Приведём описание интерфейса `IAsyncResult`:

```
interface IAsyncResult
{
    object AsyncState{ get; }
    WaitHandle AsyncWaitHandle{ get; }
    bool CompletedSynchronously{ get; }
    bool IsCompleted{ get; }
}
```

Поле `IsCompleted` позволяет узнать, завершилась ли работа асинхронного метода. В поле `AsyncWaitHandle` хранится объект типа `WaitHandle`. Программист может вызывать методы объекта `WaitHandle` для контроля над потоком выполнения асинхронного метода. Объект `AsyncState` хранит последний аргумент, указанный при вызове `BeginInvoke()`.

Делегат для функции завершения описан следующим образом:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Как видим, функции завершения передаётся единственный аргумент – объект, реализующий интерфейс `IAsyncResult`.

Рассмотрим пример асинхронного вызова метода, который вычисляет и печатает факториал целого числа. Ни функции завершения, ни возвращаемое методом `BeginInvoke()` значение не используются. Подобный подход при работе с асинхронными методами называется *«выстрелил и забыл»* (fire and forget).

```
// создадим лямбду, чтобы вычислять факториал
Func<uint, BigInteger> factorial = null;
factorial = n => (n == 0) ? 1 : n * factorial(n - 1);
// создадим лямбду, чтобы печатать факториал
Action<uint> print = n => Console.WriteLine(factorial(n));
// запустим метод асинхронно, игнорируя дополнительные параметры
print.BeginInvoke(8000, null, null);
```

```
// эмулируем работу (факториал увидим где-то на третьей итерации)
for (int i = 1; i < 10; i++)
{
    Console.Write("Do some work...");
    Thread.Sleep(3000);
}
```

Модифицируем предыдущий пример. Будем передавать в `BeginInvoke()` функцию завершения и дополнительный аргумент.

```
// объект print не изменился
// функция завершения будет выводить время выполнения метода
AsyncCallback timer = ar =>
                          {
                              var dt = (DateTime) ar.AsyncState;
                              Console.WriteLine(DateTime.Now - dt);
                          };
print.BeginInvoke(8000, timer, DateTime.Now);
print.BeginInvoke(1000, timer, DateTime.Now);
```

В разобранных примерах использовались асинхронные методы, которые не возвращают значения. В приложении может возникнуть необходимость в асинхронных методах-функциях. Для получения результата работы асинхронной функции предназначен метод `EndInvoke()`. Параметры `EndInvoke()` определяются на основе параметров метода, инкапсулированного делегатом. Во-первых, `EndInvoke()` является функцией, тип которой совпадает с типом инкапсулируемого метода. Во-вторых, метод `EndInvoke()` содержит все `out` и `ref` параметры делегата, а последний параметр имеет тип `IAsyncResult`. При вызове метода `EndInvoke()` основной поток выполнения приостанавливается до завершения работы соответствующего асинхронного метода.

Используем метод `EndInvoke()` при вычислении и печати факториала:

```
// объект factorial определён в первом примере
// так как отслеживаем окончание работы методов,
// сохраняем результат вызова BeginInvoke()
IAsyncResult ar1 = factorial.BeginInvoke(8000, null, null);
IAsyncResult ar2 = factorial.BeginInvoke(1000, null, null);
Thread.Sleep(2000);
// получаем результат второго вызова и печатаем его
BigInteger res1 = factorial.EndInvoke(ar2);
Console.WriteLine(res1);
// получаем и печатаем результат первого вызова
BigInteger res2 = factorial.EndInvoke(ar1);
Console.WriteLine(res2);
```

В заключение заметим, что технически асинхронный вызов методов реализуется средой исполнения при помощи пула потоков.

Литература

1. Албахари, Дж. C# 5.0. Справочник. Полное описание языка: Пер. с англ. / Дж. Албахари, Б. Албахари. – 5-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1008 с.: ил.

2. Нэш, Т. C# 2010: ускоренный курс для профессионалов / Т. Нэш. – М. : Издательский дом «Вильямс», 2010. – 592 с.

3. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – 6-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1312 с.: ил.

4. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж. Рихтер. – 4-е изд. – Спб.: Питер, 2013. – 896 с.: ил.

5. Фримен, А. LINQ: язык интегрированных запросов в C# 2010 для профессионалов / А. Фримен, Дж. С. Раттц-мл. – М. : Издательский дом «Вильямс», 2011. – 656 с.

6. Хейлсберг, А. Язык программирования C#. Классика Computers Science. / А. Хейлсберг, М. Торгерсен, С. Вилтамут, П. Голд. – 4-е изд. – Спб.: Питер, 2012. – 784 с.: ил.

7. Цвалина, К. Инфраструктура программных проектов: соглашения, идиомы и шаблоны для многократно используемых библиотек .NET. : Пер. с англ. / К. Цвалина. – М.: ООО «И.Д. Вильямс», 2011. – 416 с.: ил.