КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

*С. В. МАКЛЕЦОВ*

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C#

*Учебное пособие*

Казань – 2024

**УДК 004.43**

*Печатается по решению Учебно-методической комиссии*
*Института математики и механики им. Н.И. Лобачевского*
*(протокол №3 от 7.12.2023г.)*

*Рецензент: зав. кафедрой информационных систем*
*Института вычислительной математики и информационных технологий,*
*к.ф.-м.н.* ***Ф. М. Гафаров***

**Маклецов С. В.**

Объектно-ориентированное программирование на языке C#: учебное пособие / С. В. Маклецов. – Казань: Казанский (Приволжский) федеральный университет, 2024. – 132 с.

Настоящее учебное пособие предназначено для студентов-бакалавров, обучающихся по направлениям 02.03.01 – «Математика и компьютерные науки» и 01.03.01 – «Математика», а также студентов, обучающихся по направлению 09.03.02 – «Информационные системы и технологии». Пособие включает теоретические сведения по основам объектно-ориентированного программирования, разобранные примеры программ, написанных на языке программирования C#, задания для самостоятельной работы, список рекомендуемой литературы.

# ОГЛАВЛЕНИЕ

# ВВЕДЕНИЕ

Уважаемые студенты! В рамках первой части курса «Объектно-ориентированное программирование», темы которой рассматриваются в настоящем пособии, вам предлагается изучить основной язык, используемый для написания программ на платформе Microsoft .NET, – C# (си-шарп).

Для успешного освоения материала и приобретения практики программирования вам потребуется установить среду разработки MS Visual Studio версии 2022 или выше.

Настоящее пособие содержит теоретический материал, примеры программ и задания для самостоятельного выполнения для каждой из тем учебной программы.

В рамках курса изучаются следующие темы:

- основы объектно-ориентированного программирования на языке программирования C#;
- создание приложений, использующих графический интерфейс пользователя (Windows Forms);
- создание приложений с несколькими потоками; синхронизация потоков в C#;
- использование параллельных вычислений для численного решения математических задач с использованием классов библиотеки .NET.

---

Для успешного усвоения материала необходимо, чтобы читатель был знаком с базовыми принципами программирования на языке C/C++ или Java.

---

# I. ЯЗЫК ПРОГРАММИРОВАНИЯ C#

## 1. БАЗОВЫЙ СИНТАКСИС И ОСНОВЫ ООП

### Платформа .NET и основные понятия ООП

Язык программирования C# является одним из самых мощных, быстро развивающихся и востребованных языков программирования. В настоящий момент на нем пишутся самые различные приложения: от небольших программ для персональных компьютеров до крупных веб-порталов и веб-сервисов, обслуживающих ежедневно миллионы пользователей.

C# является основным языком, предназначенным для использования совместно с платформой Microsoft .NET и относится к семье языков с C-подобным синтаксисом, это означает, что правила записи программ на этом языке наиболее приближены к тем, что используются в C++ и Java. C# является объектно-ориентированным языком, имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов, делегаты, атрибуты, события, переменные, свойства, обобщённые типы и методы, итераторы, лямбда-выражения с поддержкой замыканий, LINQ, исключения.

Платформа .NET долгое время развивалась как платформа, предназначенная исключительно для работы с операционной системой Microsoft Windows, и называлась .NET Framework. Однако, в 2019 году вышла последняя версия этого варианта платформы – .NET *Framework* 4.8 и с тех пор она больше не развивается. При этом с 2014 Microsoft стала развивать её альтернативный вариант – .NET *Core*, которая уже предназначалась для разных платформ и должна была вобрать в себя все возможности устаревшего .NET Framework, добавив новую функциональность. Текущей версией является рассматриваемая в этом пособии платформа .NET 7. Именно об этой версии в совокупности с 11-ой версией языка C# будет идти речь в настоящем пособии.

C# разрабатывался как язык программирования прикладного уровня для CLR – общеязыковой средой выполнения программ, являющейся основным компонентом платформы Microsoft .NET. CLR компилирует код приложения в код на промежуточном языке (Intermediate Language – IL). После этого, на втором этапе с помощью виртуальной машины, выполняется преобразование промежуточного кода в машинный код конкретного процессора и его выполнение. Другой задачей, среды CLR является управление памятью при размещении объектов с помощью сборщика мусора (Garbage Collector). Также в функции среды входит предоставление программам доступ к библиотеке классов .NET.

Во многом возможности языка C# зависят от возможностей самой CLR. Это касается, прежде всего, системы типов C#, которая отражает систему типов BCL, используемую в .NET. Также CLR предоставляет C#, как и всем другим .NET-ориентированным языкам, многие возможности, которых лишены «классические» языки программирования. Например, сборка мусора не реализована в самом C#, а производится средой CLR.

Как уже отмечалось в самом начале, язык C# является объектно-ориентированным, а значит все программы, написанные на нем, так или иначе являются объектно-ориентированными. В связи с этим необходимо познакомиться с основными понятиями и принципами такого подхода к написанию кода.

***Объектно-ориентированное программирование*** (ООП) – это методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования. Объектно-ориентированное программирование, или сокращенно ООП, основано на таких понятиях как «абстракция», «инкапсуляция», «наследование», «полиморфизм».

***Абстракция*** – это принцип, в соответствии с которым выполняется выделение в моделируемом предмете тех его сторон, которые важны для решения конкретной задачи. В конечном счёте – абстракция представляет собой контекстное понимание предмета, и формализуется на языке программирования в форме класса.

В языке программирования ***класс*** – абстракция, представляющая собой модель некоторой информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым. В объектно-ориентированных языках программирования класс представляет собой универсальный комплексный тип данных, содержащий описание характеристик и поведения моделируемой сущности, и содержащий в себе тематически объединенный набор ***полей*** (переменных класса) и ***методов*** (функций класса).

Поскольку класс представляет собой тип данных, для использования его в программе, как правило, требуется создание переменных такого типа. Такие переменные будут представлять конкретные экземпляры объектов реального мира и, соответственно, называются ***объектами*** или ***экземплярами класса***.

Например, комплексное число может быть реализовано в форме класса, в котором будут два поля – вещественная и мнимая часть комплексного числа, а также в нем будут реализованы методы, позволяющие выполнять действия (арифметические операции) с комплексными числами. При

этом для выполнения этих действие будет необходимо использовать какие-то конкретные комплексные числа – экземпляры класса.

Наряду с полями и методами в классах также могут находиться и некоторые другие элементы. Еще одним таким, наиболее распространенным, видом элементов класса являются свойства.

***Свойство*** – это элемент класса, представляющий собой способ доступа к данным класса, имитирующий переменную некоторого типа. Обращение к свойству объекта выглядит так же, как и обращение к полю, но при этом реализовано через вызов функции-аксессора. При попытке получить значение данного свойства вызывается один метод – **`get`**, а при задании значения – другой: **`set`** либо **`init`**.

Объект с конкретными значениями полей и свойств используется через свой ***интерфейс*** – совокупность правил доступа. При этом «снаружи» имеется только информация о тех элементах класса, к которым его разработчиком был разрешен доступ. Часть элементов класса, а также детали реализации всех его методов и свойств, как правило, скрываются от пользователя.

Свойство системы, позволяющее объединять данные и действия над ними в рамках класса, а также возможность сокрытия деталей реализации называется ***инкапсуляцией***.

С целью сокращения кода, используемого для написания программы, а также для его упорядочивания, в объектно-ориентированном программировании применяется принцип наследования.

***Наследование*** – это свойство системы, позволяющее описать новый класс на основе существующего с частично или полностью заимствованной функциональностью.

Класс, от которого производится наследование, называется *базовым*, родительским или суперклассом. Новый класс – *потомком*, наследником, дочерним или производным классом. Порожденный класс (потомок), наследующий характеристики другого класса, сразу обладает теми же возможностями, что и класс-предок. При этом базовый класс остается без изменения, а классу-потомку может расширять его возможности благодаря добавлению новых элементов (полей, методов, свойств) или изменять работу некоторых методов своего родителя.

Все классы объединены в общую иерархию, на вершине которой в C# располагается базовый тип (корневой класс) **`Object`**, описанный в пространстве имен **`System`**. Он является предком для всех классов, даже если это не указывается явно. При этом все классы-потомки являются подвидами класса-предка, который содержит в себе информацию обо всех общих элементах

своих потомков, а значит, в определенном смысле является их обобщением. Таким образом, к объектам классов-потомков можно обращаться через переменную базового типа (при этом, они могут совершать различные действия). Такое поведение соответствует еще одном базовому принципу объектно-ориентированного программирования, называемому полиморфизм.

***Полиморфизм*** – это свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Однако термин полиморфизм может определяться двояко. Иногда под этим понятием понимают параметрический полиморфизм, который позволяет создавать классы, способные содержать в себе элементы не фиксированных, а произвольных типов, что также называют обобщённым программированием.

Отметим также, что язык C# является полностью объектно-ориентированным и в нем отсутствуют примитивные типы (вроде int, double или char, с которыми можно оперировать на языке C++). При этом одноименные типы для числовых данных в языке, безусловно присутствуют, только они представляют собой структуры.

*Структура* в C# своим устройством полностью совпадает с классом, она также может содержать такие элементы как поля, методы и свойства. Однако между ними есть единственное отличие. Типы-классы являются ссылочными. Это означает, что переменные этих типов на самом деле хранят адреса памяти, где находится конкретный объект. В то же время переменные типа структур хранят непосредственно сами значения.

### Базовые типы данных языка C#

В таблице I-1 представлены базовые типы, которые можно использовать в языке C#.

**Таблица I-1**

**Список базовых типов данных**

| Ключевое слово | Тип .NET | Описание |
|---|---|---|
| `object` | Object | Класс, находящийся в корне иерархии классов и являющийся предком всех классов, которые существуют или могут существовать в языке. Наиболее |

| Ключевое слово | Тип .NET | Описание |
|---|---|---|
| | | общий тип данных, способный хранить объект любого типа. |
| **bool** | Boolean | Переменные, этого класса могут содержать одно из двух возможных значений: **true** или **false**. |
| **sbyte** | SByte | 8-битное целое число со знаком.<br>Диапазон значений: от -128 до 127. |
| **byte** | Byte | 8-битное целое число без знака.<br>Диапазон значений: от 0 до 255. |
| **short** | Int16 | 16-битное целое число со знаком.<br>Диапазон значений: от -32 768 до 32 767 |
| **ushort** | UInt16 | 16-битное целое число без знака.<br>Диапазон значений: от 0 до 65 535 |
| **int** | Int32 | 32-битное целое число со знаком.<br>Диапазон значений: от $-2^{31}$ = -2 147 483 648 до $2^{31}-1$ = 2 147 483 647. |
| **uint** | UInt32 | 32-битное целое число без знака.<br>Диапазон значений: от 0 до $2^{32}-1$ = 4 294 967 295. |
| **long** | Int64 | 64-битное целое число со знаком.<br>Диапазон значений: от $-2^{63}$ = -9 223 372 036 854 775 808 до $2^{63}-1$ = 9 223 372 036 854 775 807. |
| **ulong** | UInt64 | 64-битное целое число без знака<br>Диапазон значений:<br>от 0 до $2^{64}-1$ = 18 446 744 073 709 551 615. |
| **float** | Single | Вещественное 4-байтовое число.<br>Диапазон значений: от $\pm1{,}5\cdot10^{-45}$ до $\pm3{,}4\cdot10^{38}$. Точность: 6-9 знаков. |
| **double** | Double | 8-байтовое число с плавающей точкой удвоенной точности от $\pm5{,}0\cdot10^{-324}$ до $\pm1{,}7\cdot10^{308}$. Точность: 15-17 знаков. |
| **decimal** | Decimal | Числовой десятичный формат данных |

| Ключевое слово | Тип .NET | Описание |
|---|---|---|
| | | Диапазон значения: от $\pm 1{,}0 \cdot 10^{-28}$ до $\pm 7{,}9228 \cdot 10^{28}$. Точность: 28-29 знаков. |
| `char` | Char | Символ в кодировке UNICODE. Символ может занимать в памяти 1 или 2 байта в зависимости от конкретного значения. |
| `string` | String | Строки в кодировке UNICODE. |
| `void` | Void | Тип используется при описании методов для указания того, что возвращаемое значение отсутствует. |

Для работы с данными необходимо создать переменную подходящего типа.

Ниже приведен пример объявления переменных различных типов. Обратите внимание также на использование литеральных суффиксов после неименованных констант для придания им свойств соответствующих типов.

---

**Обратите внимание!** Начиная с 9 версии языка C# у программистов появилась возможность добавлять в программы инструкции верхнего уровня, то есть действия, записанные вне рамок каких-либо методов. Такой подход удобно использовать вместо написания класса, содержащего стандартную точку входа – статический метод `Main()`. При этом стоит учитывать, что инструкции верхнего уровня, должны находиться в файле до объявления каких-либо классов, структур или функций. Таким образом, следующий код может быть записан в качестве инструкции верхнего уровня.

---

```
1   bool b = true; // объявление булевской переменной
2   sbyte b8 = 32; // использование ключевого слова в качестве названия типа
3   Byte ub8 = 255; // использование названия типа .NET в качестве названия типа
4   short sh = -1000;
5   ushort ush = 50_000; // знак подчеркивания можно использовать как разделитель разрядов
6   int dec = 255;
7   int hex = 0xff; // здесь константа записана в виде шестнадцатеричного числа (префикс 0x)
8   int bin = 0b1111_1111; // здесь константа записана в виде двоичного числа (префикс 0b)
9   // также обратите внимание, что по значению dec == hex == bin
10
11  long l = 10L; // Длинное целое (можно использовать суффикс L для констант)
12  ulong ul = 10UL; // Беззнаковое длинное целое (можно использовать суффикс UL)
13  float f = 3.5f; // Суффикс f применяется для данных типа Single
14  double d = 3.5;
```

```
15  decimal decim = 3.55m; // Суффикс m используется для значения типа decimal
16
17  // Ниже приведены различные способы задания символьных величин
18  char c1 = 'j';
19  char c2 = '\u006A';
20  char c3 = '\x006A';
21  char c4 = (char)106;
22  // c1 == c2 == c3 == c4 по значению.
23  string s = "jjjj";
```

Локальные переменные (переменные, которые описываются в теле метода) также могут быть объявлены с помощью ключевого слова **`var`**. При использовании этого ключевого слова необходимо сразу же проинициализировать переменную, тогда ее тип будет выведен по типу значения в правой части оператора присваивания.

```
1  var ns = "Привет!";
```

Так, в этом примере тип переменной **`ns`** – это **`string`**, поскольку справа от оператора присваивания располагается строковая константа.

Если после имени типа будет поставлен знак вопроса, то это будет говорить компилятору о создании обнуляемого типа. Переменные таких типов помимо всех допустимых значений из соответствующего типу диапазона, могут содержать значение **`null`**.

```
1  bool? nb = null;
2  int? ni = null;
3  //…
4  ni = 10;
```

Так, переменная **`nb`** из примера выше допускает присвоение ей любого из трех значений: **`true`**, **`false`**, **`null`**. Целочисленной переменной **`ni`** в примере также присваивается значение **`null`**. Этой возможностью удобно пользоваться, при возникновении ошибки в программе, когда невозможно получить корректный результат (например, при попытке деления на 0 в качестве результата можно возвращать **`null`**).

Тип значения можно неявно преобразовать в соответствующий обнуляемый тип значения. При использовании таких типов следует проверять их значения перед выполнением операций. Например, это можно сделать следующим образом:

```
1  int? ni = null;
2  if (ni is int i)
3      Console.WriteLine(i);
4  else
5      Console.WriteLine("No value");
```

В случаях, когда необходимо присвоить значение обнуляемого типа, переменной базового типа-значения, может потребоваться указать значение, назначаемое вместо **null**. Для этого используйте оператор «**??**».

```
6    var j = ni ?? 0;
```

Такой оператор вернет значение левого операнда, если он не равен **null**, и значение правого операнда в противном случае.

### Перечисления – enum

Перечисление в C# представляет собой расширенный аналог обычных перечислений (**enum**) языка C++. Класс-перечисление имеет простую структуру и определяет ряд именованных значений. При этом он фактически является целочисленным типом значением. Пример создания enum-класса приведен ниже.

```
1    enum Month
2    {
3        January = 1, February, March, April, May, June,
4        July, August, September, October, November, December
5        /**Класс содержит перечисление месяцев: Январь
6          * (этому элементу присваивается порядковый номер 1),
7          * Февраль (2), Март (3), Апрель (4) и т.д.
8          **/
9    }
```

Благодаря тому, что перечисление наследует от класса **Object**, в состав которого входит метод **ToString()**, у программиста есть возможность применить его для получения текстового значения элемента перечисления, что удобно, например, для вывода на экран:

```
1    Console.WriteLine((Month)10); // October
2    Console.WriteLine((Month)20); // 20
3    enum Month
4    {
5        January = 1, February, March, April, May, June,
6        July, August, September, October, November, December
7        /**Класс содержит перечисление месяцев: Январь
8          * (этому элементу присваивается порядковый номер 1),
9          * Февраль (2), Март (3), Апрель (4) и т.д.
10         **/
11   }
```

По умолчанию значения, приведенные в перечислении, представляют собой константы типа **int**, которые нумеруются по порядку, начиная с нуля. Начальное значение (см. пример выше), а также значение любой из констант всегда можно изменить, указав его явным образом. Значения последующих элементов перечисления продолжают нумероваться подряд начиная со

значения явно заданной константы. Также явно можно указать любой целочисленный тип в качестве базового через двоеточие после имени перечисления.

Также, снабдив перечисление атрибутом `[Flags]` можно разрешить присваивать переменной типа перечисления несколько значений, за счет трактовки числовых величин как битовых флагов.

```
1   PrintDay(Days.Wednesday);
2   PrintDay(Days.Monday | Days.Wednesday | Days.Friday);
3   PrintDay(Days.WorkWeek);
4   PrintDay(Days.Saturday | Days.Sunday);
5   void PrintDay(Days d)
6   {
7       Console.WriteLine(d);
8   }
9
10  [Flags]
11  enum Days : byte
12  {
13      None      = 0b_00000000,
14      Monday    = 0b_00000001,
15      Tuesday   = 0b_00000010,
16      Wednesday = 0b_00000100,
17      Thursday  = 0b_00001000,
18      Friday    = 0b_00010000,
19      Saturday  = 0b_00100000,
20      Sunday    = 0b_01000000,
21      WeekEnd   = Saturday | Sunday,
22      WorkWeek  = Monday | Tuesday | Wednesday | Thursday | Friday
23  }
```

Здесь в качестве результата работы программы будет выведено:

```
Wednesday
Monday, Wednesday, Friday
WorkWeek
WeekEnd
```

### Одномерные массивы

Как известно, массив – это набор однотипных элементов, снабженных индексами и объединенных одним именем. Индексация элементов массива в C# начинается с 0. Для создания массивов в C# используется C-подобный синтаксис. Так, например, чтобы объявить массив вещественных чисел из 10 элементов, можно написать:

```
1   int n = 10;
2   // Создание одномерного массива
3   double[] d = new double[n];
```

```
4   for (int i = 0; i < n; i++)
5   {
6       d[i] = i + 1;
7   }
```

Тем не менее, в отличие от языка C/C++, массивы в C# реализуются как объекты класса `Array`, хотя для упрощения объявления массивов упоминание этого класса можно пропускать, заменяя квадратными скобками.

Наличие в платформе .NET встроенной системы сборки мусора позволяет программисту не заботиться об освобождении памяти вручную, поскольку она будет освобождаться автоматически при ее нехватке, а также с определенным интервалом времени.

При необходимости проинициализировать массив фиксированным набором значений, можно применить инициализатор.

```
1   int[] a = {1, 10, 100, 1000}; // первичная инициализация массива
2   a = new [] {3, 4, 5, 6, 7, 8}; // замена содержимого массива новыми значениями
```

В случае выхода за границу массива, в программе произойдет исключительная ситуация, которую нужно будет либо обработать (о чем пойдет речь ниже), либо приложение аварийно завершит свою работу. Такое поведение позволяют получить бóльшую безопасность по сравнению с C/C++-программами.

Массив, как и переменную любого другого типа можно передавать в методы в качестве параметров, а также возвращать результаты типа массивов. Данные возможности продемонстрированы в следующем примере.

```
1   int[] a = {1, 10, 100, 1000}; // Инициализация массива
2   // Выведем сумму элементов массива до обработки
3   Console.WriteLine(a.Sum());
4   // Выполним преобразование массива и получим результат в новом массиве
5   var b = Sqr(a);
6   // Выведем сумму элементов нового массива
7   Console.WriteLine(b.Sum());
8
9   // Метод принимает массив в качестве параметра, а затем
10  // возвращает результат в виде нового массива
11  int[] Sqr(int[] array)
12  {
13      var res = (int[])array.Clone(); // Получение копии массива
14      for (int i = 0; i < array.Length; i++)
15      {
16          res[i] *= res[i];
17      }
18      return res;
19  }
```

Обратите внимание, что поскольку массивы являются объектами классов, в них самих уже содержатся многие полезные методы и свойства, предоставляющие удобный инструментарий для работы. Например, количество элементов массива легко узнать с помощью свойства **`Length`**. Также из приведенного фрагмента кода видно, что для подсчета, например, суммы всех элементов массива достаточно воспользоваться методом **`Sum()`**.

Еще один использованный в примере метод – **`Clone()`** позволяет получить независимую копию массива.

В результате работы приведенного кода на экране будет выведено:

```
1111
1010101
```

Некоторые полезные методы для работы с массивами можно найти в классе **`Array`**. С их помощью, в частности, можно выполнить поиск элементов в массиве, сортировку или реверсирование последовательности элементов, копирование всех или заданного диапазона элементов из одного массива в другой.

---

ПОДСКАЗКУ ПО ИМЕЮЩИМСЯ В КЛАССЕ МЕТОДАМ ЛЕГКО ПОЛУЧИТЬ, НАБРАВ ПОСЛЕ ИМЕНИ ОБЪЕКТА ИЛИ КЛАССА (В ЗАВИСИМОСТИ ОТ ВИДА МЕТОДА) СИМВОЛ ТОЧКИ.

---

```
1   int[] mas = { 6, 2, 4, 10, 8 };
2   ShowArrayElements(mas); // 6 2 4 10 8
3   Array.Sort(mas); // Сортировка массива
4   ShowArrayElements(mas); // 2 4 6 8 10
5   Array.Reverse(mas); // Изменение порядка элементов на противоположный
6   ShowArrayElements(mas); // 10 8 6 4 2
7
8   //Метод позволяет отобразить содержимое массива на экране
9   void ShowArrayElements(int[] arr)
10  {
11      foreach (var elem in arr)
12      {
13          Console.Write("{0} ", elem);
14      }
15      Console.WriteLine();
16  }
```

В приведенном выше примере можно заметить использование особой разновидности цикла – **`foreach`**, позволяющей перебирать все элементы некоторой коллекции. Работа с этим циклом имеет ряд особенностей.

1. В приведенном выше примере, при выполнении цикла, переменная **`elem`** поочередно принимает все значения из массива **`arr`**. При этом изменить эту переменную нельзя. Таким образом, данный вид цикла

можно использовать, например, для вывода элементов массива, однако он не подходит для изменения элементов.

2. При работе такого цикла состав элементов перебираемой коллекции элементов не должен изменяться, в противном случае это приведет к ошибке времени выполнения.

### Многомерные массивы

Многомерные массивы – это массивы, которые имеют более одного измерения. Среди них в программах чаще всего находят применение двумерные массивы, которые можно использовать, например, для хранения матриц. При этом каждый элемент массива имеет два индекса (номер строки и номер столбца).

Многомерные массивы в C# бывают двух видов: прямоугольные и ступенчатые.

Прямоугольный массив всегда содержит одинаковое количество столбцов для каждой строчки. Для его объявления можно использовать конструкции одного из следующих видов.

```
1   int rows = 3; // Количество строк
2   int cols = 4; // Количество столбцов
3   double[,] mas2_1 = new double[rows, cols];
4   double[,] mas2_2 = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 0, 1, 2 } };
5   mas2_2 = new double[,] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

При необходимости обращения к элементам такого массива, их индексы перечисляются через запятую в рамках одних квадратных скобок.

```
6
7   for (int i = 0; i < rows; i++)
8   {
9       for (int j = 0; j < cols; j++)
10      {
11          mas2_1[i, j] = mas2_1.GetLength(1) * i + j + 1;
12      }
13  }
```

Свойство `Length` для двумерного массива возвращает общее количество его элементов, а для получения числа элементов по какой-то одной оси (по строкам или по столбцам), можно использовать метод `GetLength()`, который в качестве параметра принимает номер измерения (0 – для строк, 1 – для столбцов).

Для работы с прямоугольными массивами также имеется возможность использовать цикл foreach, однако этот вид цикла будет перебирать все элементы, как если бы они были расположены в рамках одной строки.

```
14  int cnt = 0;
15  foreach (var elem in mas2_1)
16  {
17      Console.Write("{0,2} ", elem);
18      if (++cnt % mas2_1.GetLength(1) == 0)
19          Console.WriteLine();
20  }
```

При дополнительном контроле мест перехода на новую строку (при помощи условного оператора), на экране появится следующий вывод.

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

Таким образом, из последнего примера видно, что управляемый многомерный прямоугольный массив C#, в отличие от стандарта языка C++, представляет собой не массив массивов, а массив непосредственно самих элементов, снабженных несколькими индексами.

Однако C# позволяет создавать и другой вид массивов – ступенчатые, которые как раз представляют собой массивы массивов. Работа с ними строится по тем же принципам, что и в C/C++, что проявляется и в синтаксисе.

```
1   int r = 3; // Определяем количество строк
2   double[][] treug = new double[r][]; // Создаем массив массивов
3   int cntr = 0; // Счётчик элементов
4   for (int i = 0; i < r; i++)
5   {
6       treug[i] = new double[i + 1]; // Создаем элемент - массив
7       for (int j = 0; j < i + 1; j++)
8       {
9           treug[i][j] = ++cntr; // Заполняем элементы массива в массиве
10      }
11  }
12
13  foreach (var row in treug) // Перебирает строки
14  {
15      foreach (var elem in row) // Перебирает элементы в строке
16      {
17          Console.Write("{0, 3}", elem);
18      }
19      Console.WriteLine();
20  }
```

Данный алгоритм приведет к появлению на экране следующего результата.

```
  1
  2  3
  4  5  6
```

Ступенчатые массивы имеет смысл применять только ради экономии памяти, когда заранее известно, что в массиве будет использовано разное число элементов в разных строках. Например, в случае, когда нужно хранить только часть матрицы (верхне/нижне-треугольную и т. п.). В других ситуациях предпочтение стоит отдавать прямоугольным массивам, работа с которыми, вероятнее всего, будет производиться быстрее за счет более компактного расположения всех элементов в памяти.

### Значения типов по умолчанию

Язык C# автоматически инициализирует все поля и свойства классов значениями по умолчанию. Если создается переменная базового типа-значения, то по умолчанию она, вероятнее всего, будет равна `0` (или `0.0`). Для перечисления некоторого типа `E` значением по умолчанию будет являться `(E)0`. Для ссылочных типов значением по умолчанию является `null`.

Проинициализировать локальную переменную значением по умолчанию можно с помощью оператора `default`:

```
1  var k = default(int);
2  int k2 = default;
```

## 2. КЛАССЫ И ОБЪЕКТЫ В C#-ПРОГРАММАХ

### Уровень доступа к элементам классов

Класс (как и структура) представляет собой составной тип данных, который может содержать в себе элементы различных видов. Следующая схема содержит перечисление разновидностей членов класса.


Рис. 1. Разновидности элементов класса в C#

Основными, наиболее часто встречающимися компонентами классов в языке C# являются поля, методы и свойства. К этим, а также прочим элементам класса, может быть предоставлен различный уровень доступа. По умолчанию (если иное не указано явным образом), доступ к элементу ограничен тем классом, в котором он описан. Однако у программиста есть возможность с помощью **модификатора доступа** определить необходимую область видимости элемента, то есть указать, где будет возможно использование того или иного члена класса.

В языке C# можно использовать 4 различных вида модификаторов в 6 комбинациях. Ниже перечислены возможные комбинации, в порядке возрастания уровня доступа.

- `private` – закрытый член класса, то есть недоступный нигде больше, кроме того класса, в котором он описан. Именно такой модификатор подразумевается по умолчанию.
- `private protected` – компонент класса с этим модификатором доступен не только в классе где описан, но также и в производных от него классах, но только если они находятся в рамках того же самого проекта (той же сборки).
- `protected` – член класса доступен из самого класса и любых его наследников, вне зависимости от того, где эти наследники располагаются.
- `internal` – элемент класса с таким модификатором доступен в любом месте кода в рамках текущего проекта (сборки).
- `protected internal` – пара модификаторов разрешает доступ к элементу из любого места кода текущей сборки, а также из наследников данного класса, вне зависимости от места их расположения.
- `public` – задает самую широкую области видимости. Компонент является общедоступным как в текущей, так и любых других сборках.

---

Указанные выше уровни могут применяться по отношению как к членам классов, так и к членам структур. Однако в C# для структур существует ограничение: они не наследуются. Поэтому в них не применяются модификаторы доступа, содержащие слово `protected`.

---

Модификаторы доступа также могут указываться рядом с объявлением класса или структуры. При этом, если они не являются частью какого-либо другого класса, то для них можно использовать только модификаторы `internal` (это значение используется по умолчанию для классов размещенных в пространстве имен, если не указано явно) или `public`. В 11 версии языка

C# вводится еще один модификатор доступа, применяемый к классам, структурам, перечислениям и некоторым другим компонентам – **`file`**. Класс, объявленный с таким модификатором будет доступен только в рамках того файла, в котором находится.

В таблице I-2 наглядно отображается возможность доступа из различных мест кода к элементу класса в зависимости от присвоенного уровня доступа.

**Таблица I-2**

**Действие модификаторов доступа**

| **Место в коде** | **Модификатор уровня доступа** | | | | | | |
|---|---|---|---|---|---|---|---|
| | *отсутствует* | `private` | `private protected` | `protected` | `internal` | `protected internal` | `public` |
| Данный класс | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Производный класс в текущем проекте | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Производный класс в другом проекте | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ |
| Непроизводный класс в текущем проекте | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Непроизводный класс в другом проекте | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |

### Создание классов

Подавляющее большинство C#-программ, за исключением, пожалуй, лишь самых примитивных, пишется с использованием классов. Для объявления класса используется ключевое слово **`class`**, за которым следует имя класса.

Ниже приведен пример, в котором создается класс **`Person`**, предназначенный для хранения информации о человеке (person.cs).

```
1  namespace Example1;
2
3  /// <summary>
4  /// Объявление общедоступного класса Person
5  /// При именовании классов принято использовать Pascal-нотацию
```

```
6   /// (Каждое слово названия пишется с заглавной буквы)
7   /// </summary>
8   public class Person
9   {
10      /// <summary>
11      /// Ниже описано приватное поле класса.
12      /// Имена полей обычно начинают с маленьких букв, но если
13      /// название составное, остальные части названия пишутся с заглавных букв
14      /// (так называемый camelCase).
15      /// Перед названием приватного поля также обычно ставится знак подчеркивания.
16      /// Поля класса инициализируются значениями по умолчанию автоматически.
17      /// </summary>
18      private int _personId;
19
20      /// <summary>
21      /// Ниже описано автореализуемое свойство "Имя",
22      /// позволяющее хранить, а также задавать значения для имени.
23      /// Свойство доступно для чтения из любого места программного кода,
24      /// однако задать новое значение для имени позволено только из самого класса.
25      /// </summary>
26      public string Name { get; private set; } = "Иван Иванов";
27
28      /// <summary>
29      /// Приватное поле возраст.
30      /// Поле инициализируется значение 18; операция выполняется перед
31      /// запуском любого из конструкторов.
32      /// </summary>
33      private int _age = 18;
34
35      /// <summary>
36      /// Ниже описано свойство для доступа к полю _age.
37      /// При этом, свойство выполняет проверку присваемого значения и,
38      /// при необходимости, приводит его к диапазону от 0 до 150
39      /// </summary>
40      public int Age
41      {
42          get => _age;
43          set => _age = Math.Max(Math.Min(value, 150), 0);
44      }
45
46      /// <summary>
47      /// Вычисляемое свойство только для чтения, позволяющее получить
48      /// некоторую сводную информацию о персоне в виде строки.
49      /// </summary>
50      public string PersonData
51      {
52          get => Name + ". (" + Age + ").";
53      }
54
55      /// <summary>
56      /// Свойство для получения идентификатора человека, а также
```

```
57          /// для его инициализации.
58          /// </summary>
59          public int PersonId
60          {
61              get => _personId;
62              init => _personId = value;
63          }
64
65          /// <summary>
66          /// Конструктор класса по умолчанию. Несмотря на то, что конструктор пуст, его необходимо
67          /// объявить явно, чтобы иметь возможность использовать, поскольку в классе есть и другие
68          /// реализации конструкторов
69          /// </summary>
70          public Person() {}
71
72          /// <summary>
73          /// Перегруженная версия конструктора класса с параметрами
74          /// Параметры методов класса принято начинать с маленькой буквы
75          /// и использовать camelCase.
76          /// </summary>
77          public Person(int id, string name, int age = 18)
78          {
79              // В этом методе использовано ключевое слово this, которое позволяет
80              // указать на объект класса, для которого в данный момент исполняется код.
81              // При отсутствии конфликта имен, данное ключевое слово можно пропускать.
82              _personId = id;
83              this.Name = name;
84              Age = age;
85          }
86
87          /// <summary>
88          /// Метод, предоставляющий полную информацию о человеке в формате строки
89          /// </summary>
90          /// <returns>
91          /// Строка, содержащая развернутую информацию о человеке
92          /// </returns>
93          public override string ToString() => _personId + ". " + PersonData;
94
95          /// <summary>
96          /// Метод, позволяющий определить идентичность двух персон по их данным
97          /// (применение обычной операции "==", как правило,
98          /// будет приводить к сравнению ссылок на экземпляры классов, а не сравнению значений)
99          /// </summary>
100         /// <param name="obj">Объект, с которым будет сравниваться текущий экземпляр</param>
101         /// <returns>true, если объекты обладают одинаковыми свойствами и
102         /// false в противном случае</returns>
103         public override bool Equals(object? obj)
104         {
105             return obj is Person person2
106                    && Name == person2.Name
```

```
107                 && Age == person2.Age;
108     }
109
110     /// <summary>
111     /// При переопределении метода Equals также принято реализовывать в классе
112     /// переопределенный метод GetHashCode для корректной работы некоторых коллекций
113     /// </summary>
114     /// <returns>Хэш код, полученный для экземпляра класса</returns>
115     public override int GetHashCode() => HashCode.Combine(Name, Age);
116 }
```

**Структуры.** Помимо классов в C# у программиста есть возможность создать тип-структуру (`struct`). Структуры, в отличие от классов, хранят непосредственно значения (классы же являются представителями ссылочных типов, то есть работать с ними следует как с указателями в C++). Кроме того, отличием структур от классов является то, что они не образуют иерархии. То есть для структур нет наследования (структура не может являться базой для построения другой структуры или класса, а также не может являться наследником класса).

## Конструкторы класса

Для работы с классом, который описан в примере выше, нужно будет создать переменную этого типа – экземпляр или объект класса. Для этого нужно воспользоваться одним из конструкторов.

```
1  using Example1;
2
3  // Инициализация объекта p1 конструктором по умолчанию
4  Person p1 = new Person();
5  // Инициализация объекта p2 конструктором с параметрами
6  var p2 = new Person(1, "Петр Васечкин", 20);
7  // Инициализация объекта p3 конструктором с параметром с дальнейшим
8  // применением инициализатора свойств
9  var p3 = new Person(2, "Адель Хабибуллин"){ Age = 19 };
```

Здесь созданы три объекта: `p1`, `p2` и `p3`. Для создания первого из них был использован конструктор по умолчанию, а для второго – его перегруженная версия, с параметрами. Третий создан с использованием того же конструктора, в котором один из параметров конструктора опущен, благодаря использованию значения по умолчанию. Однако затем значение `18` для возраста заменяется значением `19` при помощи применяемого к вновь создаваемому объекту класса инициализатора, в котором указывается новое значение возраста.

Напомним, что ***конструктором*** называется метод класса, который вызывается при создании его экземпляров и служит для инициализации объекта. В языке C# конструктор называется точно так же, как сам класс и не

имеет типа возвращаемого значения. (Совсем. Не указывается даже слово **`void`**).

---

**НЕСКОЛЬКО ВАЖНЫХ ЗАМЕЧАНИЙ**

ЕСЛИ В КЛАССЕ НЕ ОБЪЯВЛЕН НИ ОДИН КОНСТРУКТОР, ТО В НЕМ БУДЕТ АВТОМАТИЧЕСКИ СОЗДАН КОНСТРУКТОР ПО УМОЛЧАНИЮ, КОТОРЫЙ ИНИЦИАЛИЗИРУЕТ ВСЕ ПОЛЯ КЛАССА ЗНАЧЕНИЯМИ ПО УМОЛЧАНИЮ.

ЕСЛИ В КЛАССЕ ЕСТЬ ХОТЯ БЫ ОДИН КОНСТРУКТОР, ОПРЕДЕЛЕННЫЙ ПРОГРАММИСТОМ, ТО КОНСТРУКТОР ПО УМОЛЧАНИЮ НЕ ДОБАВЛЯЕТСЯ, ДАЖЕ ЕСЛИ ПРОГРАММИСТ ЕГО НЕ СОЗДАЛ.

ВСЕ ПОЛЯ И СВОЙСТВА КЛАССА ИНИЦИАЛИЗИРУЮТСЯ ЗНАЧЕНИЯМИ ПО УМОЛЧАНИЮ. ПОЭТОМУ ДЛЯ ПРИДАНИЯ, НАПРИМЕР, НУЛЕВЫХ ЗНАЧЕНИЙ ЧИСЛОВЫМ ПЕРЕМЕННЫМ, ИЛИ NULL-ЗНАЧЕНИЙ ДЛЯ ПЕРЕМЕННЫХ КЛАССОВ ОТДЕЛЬНЫХ ДЕЙСТВИЙ В КОНСТРУКТОРЕ НЕ ТРЕБУЕТСЯ.

---

Обратите внимание, что в классе как правило доступна ссылка на объект, для которого в данный момент выполняется код метода или свойства через ключевое слово **`this`**. Зачастую это слово используется для разрешения конфликта имен, когда, например, в конструкторе нужно присвоить значение параметра одноименному полю.

### СВОЙСТВО КАК ЭЛЕМЕНТ КЛАССА

Свойство – это особый элемент класса, который может оказаться более сложным для понимая, чем поле (переменная класса) или метод (функция класса). Поэтому есть смысл остановиться на этом виде элементов более подробно.

С одной стороны, при работе с классом имеется возможность обратиться к его свойству как к полю, присвоив ему какое-либо значение (**`set`**), или наоборот, получив некоторое значение, хранящееся или вычисляемое в объекте класса (**`get`**).

```
10  Console.WriteLine(p1.Age); // Получение возраста человека через свойство Age (18)
11  p2.Age = 22; // Обновление возраста через свойство
```

Такое использование внешне ничем не отличается от работы с общедоступными полями.

С другой стороны, свойства позволяют выполнять определенный набор действий перед заданием или чтением значений, что роднит их с методами.

Так, в приведенном выше примере свойство **Age** выполняет проверку присваиваемого значения и не дает ему выйти за границы диапазона от 0 до 150.

```
12  p1.Age = -100;
13  Console.WriteLine(p1.Age); //0
```

Кроме того, при помощи свойств можно открывать избирательный доступ к полю. Например, для того, чтобы сделать автореализуемое свойство **Name** доступным только для чтения вне класса, его метод-аксессор **set** был помечен как приватный.

---

ОТМЕТИМ, ЧТО ***АВТОРЕАЛИЗУЕМЫМ*** НАЗЫВАЕТСЯ СВОЙСТВО, У КОТОРОГО ПОСЛЕ МЕТОДА(ОВ)-АКСЕССОРОВ СРАЗУ СТАВИТСЯ ТОЧКА С ЗАПЯТОЙ. В ЭТОМ СЛУЧАЕ ДЛЯ СВОЙСТВА СОЗДАЕТСЯ ТЕНЕВОЕ ПОЛЕ (BACKING FIELD), В КОТОРОЕ ПОМЕЩАЕТСЯ, И ИЗ КОТОРОГО ИЗВЛЕКАЕТСЯ ЗНАЧЕНИЕ, ДОСТУП К КОТОРОМУ ОРГАНИЗУЕТСЯ ЧЕРЕЗ УКАЗАННОЕ СВОЙСТВО.

АВТОРЕАЛИЗУЕМОЕ СВОЙСТВО МОЖЕТ БЫТЬ ПРОИНИЦИАЛИЗИРОВАНО ЗНАЧЕНИЕМ ПОСЛЕ ЗАКРЫВАЮЩЕЙСЯ ФИГУРНОЙ СКОБКИ.

---

Если же, например, требуется сделать свойство доступным только для чтения из любого места кода, в том числе из самого класса, метод **set** из него можно полностью удалить, как это сделано для вычисляемого свойства **PersonData**. Аналогичным образом из свойства можно изъять метод **get**, для того чтобы его можно было использовать только для задания значений, но не для их получения. Модификатор доступа может быть задан только для одного из методов-аксессоров, но не для обоих сразу.

Вместо метода-аксессора **set** в свойствах может располагаться метод **init**, как, например, это реализовано в свойстве **PersonId** примера. Он позволяет проинициализировать значение свойства, задав его значение в рамках конструктора. Во всех остальных местах кода, присвоение значения свойству окажется невозможным.

### Индексаторы

В языке C# у программиста есть возможность создать одно особое свойство, с именем **this**, за которым ставятся квадратные скобки с параметрами. Описанное таким образом свойство называется индексатором и может быть использовано для получения доступа к элементам класса по индексу.

Добавим к предыдущему примеру с классом **Person**, еще один класс – **Group**, содержащему список людей, объединенных в общую группу (group.cs).

```
1   namespace Example1;
2
3   /// <summary>
4   /// Класс для работы с группой людей
5   /// </summary>
6   public class Group
7   {
8       /// <summary>
9       /// Идентификатор группы
10      /// </summary>
11      public int Id { get; set; }
12
13      /// <summary>
14      /// Массив персон
15      /// </summary>
16      private Person[] _persons;
17      /// <summary>
18      /// Конструктор класса, позволяющий создать новую группу людей
19      /// </summary>
20      /// <param name="id">Идентификатор группы</param>
21      /// <param name="persons">Список персон</param>
22      public Group(int id, params Person[] persons)
23      {
24          Id = id;
25          // Создание копии списка персон
26          _persons = new Person[persons.Length];
27          Array.Copy(persons, _persons, persons.Length);
28      }
29
30      /// <summary>
31      /// Индексатор
32      /// </summary>
33      /// <param name="serialNum">Порядковый номер человека в группе</param>
34      /// <returns>Возвращает или задает персону
35      /// по ее порядковому номеру в группе</returns>
36      public Person? this[int serialNum]
37      {
38          get
39          {
40              if (serialNum >= 0 && serialNum < _persons.Length)
41                  return _persons[serialNum];
42              return null;
43          }
44          set
45          {
46              if (serialNum >= 0 && serialNum < _persons.Length)
47                  _persons[serialNum] = value;
48          }
49      }
50  }
```

Данный класс содержит индексатор, с помощью которого можно получить информацию о конкретном члене группы или изменить его.

Для этого можно написать следующий код в файле main.cs.

```
14  var gr = new Group(1, p1, p2, p3);
15  // Получаем доступ к члену группы с номером 0
16  gr[0].Age = 22;
17  Console.WriteLine(gr[0]);
18  // Изменяем члена группы с номером 0.
19  gr[0] = new Person(5, "Николай Сидоров", 30);
20  Console.Write(gr[0]);
```

На экране будет отображаться.

```
0. Иван Иванов. (22).
5. Николай Сидоров. (30).
```

## Методы

Как отмечалось ранее, метод – это функция, описанная в классе. В целом, запись методов в языке C# похожа на аналогичные конструкции в таких языках программирования как C++ или Java. Однако при объявлении и вызове методов на C# есть и некоторые отличительные особенности.

### *Сокращенная запись методов*

Если тело метода содержит единственное выражение, то запись такого метода можно сократить. Так, например, рассмотрим метод вычисляющий квадрата числа.

```
1  double sqr(double x)
2  {
3      return x * x;
4  }
```

Поскольку его тело содержит единственное выражение, такой метод можно переписать в краткой форме.

```
1  double sqr(double x) => x * x;
```

### *Именованные параметры*

Рассмотрим метод, принимающий несколько параметров.

```
1  void ShowInfo(string name, float salary)
2  {
3      Console.WriteLine(
4          $"Зарплата работника {name} составляет {salary}р."
5      );
6  }
```

Для его вызова параметры можно передать в том порядке, который указан при объявлении.

```
7    ShowInfo("Иванов И. И.", 50000f);
```

Однако, при желании программист может использовать при вызове иной порядок параметров, указывая их названия.

```
7    ShowInfo(salary: 50000f, name: "Иванов И. И.");
```

***Передача параметров по ссылке и использование выходных параметров***

Если в некоторый метод нужно передать параметр, изменить его там и воспользоваться изменённым значением в вызвавшем методе, параметр нужно передавать по ссылке. Однако ссылочными типами в C# являются только представители классов. Если же тип является структурой, таким, например, как `int` или `double`, то передача такого параметра будет выполняться по значению. Для изменения этого поведения можно воспользоваться способом передачи параметра по ссылке.

Следующий фрагмент кода содержит метод, принимающий параметры по значению.

```
1    void Swap(int a, int b)
2    {
3        var t = a;
4        a = b;
5        b = t;
6    }
7
8    int a = 3;
9    int b = 5;
10   Console.WriteLine("a = {0}, b = {1}", a, b);
11   Swap(a, b);
12   Console.WriteLine("a = {0}, b = {1}", a, b);
```

Очевидно, что в результате на экран будет выведено следующее.

```
a = 3, b = 5
a = 3, b = 5
```

Для передачи параметров по ссылке следует добавить к ним при объявлении и вызове метода ключевое слово `ref`.

```
1    void Swap(ref int a, ref int b)
2    {
3        var t = a;
4        a = b;
5        b = t;
6    }
7
8    int a = 3;
```

```
9   int b = 5;
10  Console.WriteLine("a = {0}, b = {1}", a, b);
11  Swap(ref a, ref b);
12  Console.WriteLine("a = {0}, b = {1}", a, b);
```

Теперь на экран будут выводиться уже измененные значения.

```
a = 3, b = 5
a = 5, b = 3
```

Если метод генерирует новые значения для параметра, то такой параметр можно указать в качестве выходного (**`out`**). Такой подход удобно использовать для возврата значения из метода через параметр, особенно, когда требуется вернуть более одного значения.

```
1   void FillValues(out int a, out int b)
2   {
3       var r = new Random((int)DateTime.Now.Ticks);
4       a = r.Next(1, 101);
5       b = r.Next(1, 101);
6   }
7
8   int a, b;
9   FillValues(out a, out b);
10  Console.WriteLine($"a = {a}, b = {b}");
```

Здесь на экран будут выведены случайные значения, присвоенные переменным **`a`** и **`b`**.

```
a = 30, b = 55
```

---

**Обратите внимание!** Если вы используете выходные параметры, то обязаны присвоить им значения внутри функции. При этом до вызова функции такие параметры определять не обязательно.

---

Помимо выходных (**`out`**) параметров в C# существуют и входные (**`in`**) параметры. Их использование обеспечивает передачу параметров по ссылке, однако при этом запрещает их изменение внутри метода. В некоторых случаях передача данных по ссылке может повысить производительность программы, поскольку не будет требовать создания копии объекта (при переносе значений из фактических параметров метода в формальные), при этом in-параметр гарантирует неизменность переданного значения внутри метода.

```
1   int Calc(in int a, in int b)
2   {
3       // a++; - Если раскомментировать, то здесь будет ошибка,
4       // поскольку параметр a неизменяемый
5       return a + b;
6   }
7   Console.WriteLine(Calc(3, 5)); // 8
```

### *Методы с переменным количеством параметров*

Язык программирования C# позволяет передавать в метод произвольное количество параметров, используя для этого ключевое слово **`params`**. Параметр, перед которым стоит такое ключевое слово обязан представлять собой одномерный массив.

Несмотря на то, что количество фактических параметров в этой ситуации может быть произвольным (в том числе нулевым), их типы обязаны соответствовать указанному в параметре массиву.

```
1  int Sum(params int[] values) => values.Sum();
2  Console.WriteLine(Sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)); //55
3  Console.WriteLine(Sum(1, 2)); //3
4  Console.WriteLine(Sum()); //0
```

Также такой метод позволяет сразу передавать в него массив.

```
5  Console.WriteLine(Sum(new int[] { 1, 2, 3, 4 })); // 10
```

---

**Обратите внимание!** Параметр, снабженный ключевым словом params, обязан стоять в списке параметров последним.

---

### *Локальные функции*

Локальными называются функции, которые описаны в других функциях. Как правило, они содержат действия, которые применяются только в рамках ее метода.

Пусть требуется создать метод, который выполняет сравнение двух массивов. При этом операция сравнения определена следующим образом: массив **`a`** больше массива **`b`** в случае, когда сумма элементов массива **`a`** больше суммы элементов массива **`b`**.

Для сравнения массивов была написан метод **`Compare()`**. Он принимает два параметра и возвращает «1», если первый аргумент больше второго, «-1», если первый аргумент меньше второго и «0», если два массива равны (суммы их элементов совпадают).

```
1  static int Compare(int[] arr1, int[] arr2)
2  {
3      var s1 = 0;
4      for (int i = 0; i < arr1.Length; i++)
5      {
6          s1 += arr1[i];
7      }
8
9      var s2 = 0;
10     for (int i = 0; i < arr2.Length; i++)
11     {
12         s2 += arr2[i];
```

```
13      }
14
15      return Math.Sign(s1 - s2);
16  }
```

Можно заметить, что здесь пришлось дважды повторить запись нахождения суммы. Чтобы избавиться от повторов, можно оформить это действие в качестве локальной функции. Тогда код преобразуется следующим образом.

```
1   static int Compare(int[] arr1, int[] arr2)
2   {
3       return Math.Sign(Sum(arr1) - Sum(arr2));
4       // локальная функция:
5       int Sum(int[] arr)
6       {
7           var s = 0;
8           for (int i = 0; i < arr.Length; i++)
9           {
10              s += arr[i];
11          }
12          return s;
13      }
14  }
```

Пример использования метода.

```
15  var a = new[] { 1, 2, 3, 4, 5 };
16  var b = new[] { 10, 20, 30 };
17  Console.WriteLine(Compare(a, b)); // -1
```

### Статические члены класса

Класс, как уже отмечалось ранее, является типом данных. Обычно каждый экземпляр класса содержит свою копию данных – членов этого класса. Однако некоторые элементы требуют наличия к ним совместного доступа из всех объектов класса. Такие совместно используемые данные можно описать как часть класса, а не его объектов. Для этого применяется ключевое слово **`static`**.

Статические поля класса будут хранить общую информацию для всех его объектов.

Статические методы и свойства класса также могут быть объявлены в классе, но они смогут получить доступ только к другим статическим элементам, поскольку в них не будет доступна ссылка на конкретный объект класса (**`this`**).

Если класс содержит исключительно только статические элементы, то весь класс может быть объявлен статическим, для чего перед словом **`class`** нужно будет также поставить слово **`static`**.

Статические члены класса могут быть использованы, например, для создания счетчика объектов класса, одновременного доступа ко всем объектам или определенной их совокупности, либо для разделения всеми объектами общих ресурсов.

Ниже приведен пример класса, который выполняет подсчет количества своих экземпляров и присвоения им порядковых номеров.

```
1   namespace Example2;
2   /// <summary>
3   /// Класс, выполняющий подсчет своих экземпляров и присвоение им порядковых номеров
4   /// </summary>
5   public class ObjectCounter
6   {
7       /// <summary>
8       /// Статическое поле для хранения счётчика экземпляров
9       /// </summary>
10      private static int _counter;
11      /// <summary>
12      /// Статическое свойство для получения значения счётчика
13      /// </summary>
14      public static int ObjectCount => _counter;
15      /// <summary>
16      /// Идентификатор объекта (порядковый номер)
17      /// </summary>
18      private int _id;
19      /// <summary>
20      /// Свойство для получения данных о порядковом номере объекта
21      /// </summary>
22      public int Id => _id;
23      /// <summary>
24      /// Конструктор класса. Выполняет увеличение счётчика объектов и присвоение
25      /// номера новому экземпляру
26      /// </summary>
27      public ObjectCounter()
28      {
29          _id = ++_counter;
30      }
31      /// <summary>
32      /// Получение строкового представления объекта.
33      /// </summary>
34      /// <returns>Строка, содержащая информацию о номере объекта
35      /// и общем количестве созданных экземпляров класса</returns>
36      public override string ToString() =>
37          $"Это {Id} объект класса из {ObjectCount} созданных";
38  }
```

Протестировать работу класса можно на следующем примере.

```
1   var obj1 = new ObjectCounter();
2   Console.WriteLine(obj1); // Это 1 объект класса из 1 созданных
3   var obj2 = new ObjectCounter();
```

```
4   var obj3 = new ObjectCounter();
5   Console.WriteLine(obj3); // Это 3 объект класса из 3 созданных
6   Console.WriteLine(obj2); // Это 2 объект класса из 3 созданных
```

Информация, выводимая на экране при запуске программы приведена в комментариях, рядом с соответствующими строчками.

### Определение операторов

В некоторых случаях бывает удобно производить операции с объектами классов, используя для этого не обычные методы, а знаки операторов, так как мы, например, это делаем с числами. Язык C# позволяет выполнить перегрузку некоторых своих операторов так, чтобы они работали с объектами новых классов.

Определить в классе можно операторы, указанные в таблице I-3.

**Таблица I-3**

**Список переопределяемых операторов в языке C#**

| Тип оператора | Список операторов |
|---|---|
| Унарные операторы | +,-, !, ~, ++, --, true, false |
| Бинарные операторы | +, -, *, /, % |
| Операции сравнения | ==. !=. <=, >=, <, > |
| Поразрядные операторы | &, \|, ^, <<, >> |
| Логические операторы | &&, \|\| |

**Обратите внимание!** Операторы сравнения всегда перегружаются соответствующими парами: `==` и `!=`, `<` и `>`, `<=` и `>=`.

**Обратите внимание!** Такие операторы, как, например, оператор присваивания (в отличие от языка C++) или тернарный оператор перегрузить в C# нельзя.

Для определения в классе оператора, необходимо создать статический метод с необходимым числом параметров (их количество должно соответствовать числу операндов для оператора). Название такого метода будет содержать ключевое слово **`operator`**, за котором должен следовать знак оператора.

Еще одним видом операторов, которые можно описать в классе, являются явные (**`explicit`**) и неявные (**`implicit`**) операторы преобразования типа. С их помощью можно выполнять преобразования между различными типами данных, при этом следует учитывать, что не допускается выполнение преобразования между родственными классами (нельзя приводить тип родительского класса к потомку и наоборот).

В следующем примере описан класс для работы с комплексными числами, в котором, в том числе, реализованы операторы для выполнения арифметических операций с комплексными числами, получения сопряженного комплексного числа, сравнения чисел на равенство и неравенство, а также операторы явного и неявного приведения типа между комплексными и вещественными числами.

Файл Complex.cs.

```
1   using System.Text;
2
3   namespace Operators;
4
5   public class Complex
6   {
7       /// <summary>
8       /// Вещественная часть комплексного числа
9       /// </summary>
10      public double Re { get; set; }
11
12      /// <summary>
13      /// Мнимая часть комплексного числа
14      /// </summary>
15      public double Im { get; set; }
16
17      /// <summary>
18      /// Конструктор по умолчанию. Создает число равное 0
19      /// </summary>
20      public Complex(){}
21
22      /// <summary>
23      /// Конструктор для создания комплексного числа с заданными
24      /// вещественной и мнимой частями
25      /// </summary>
26      /// <param name="re">Вещественная часть числа</param>
27      /// <param name="im">Мнимая часть числа</param>
28      public Complex(double re, double im = 0.0)
29      {
30          Re = re;
31          Im = im;
32      }
33
34      /// <summary>
35      /// Конструктор копирования
36      /// </summary>
37      /// <param name="other">Комплексное число,
38      /// копию которого нужно создать</param>
39      public Complex(Complex other)
40      {
41          Re = other.Re;
```

```
42          Im = other.Im;
43      }
44
45      /// <summary>
46      /// Бинарный оператор. Сложение комплексных чисел
47      /// </summary>
48      /// <param name="z1">Первое слагаемое</param>
49      /// <param name="z2">Второе слагаемое</param>
50      /// <returns>Сумма комплексных чисел</returns>
51      public static Complex operator+(Complex z1, Complex z2)
52      {
53          return new Complex(z1.Re + z2.Re, z1.Im + z2.Im);
54      }
55
56      /// <summary>
57      /// Унарный оператор "+" используется для симметрии с унарным минусом
58      /// </summary>
59      /// <param name="z1">Комплексное число</param>
60      /// <returns>Копия комплексного числа из параметра</returns>
61      public static Complex operator +(Complex z1)
62      {
63          return new Complex(z1);
64      }
65
66      /// <summary>
67      /// Бинарный оператор вычитания
68      /// </summary>
69      /// <param name="z1">Первое комплексное число</param>
70      /// <param name="z2">Второе, вычитаемое, комплексное число</param>
71      /// <returns>Разность комплексных чисел</returns>
72      public static Complex operator -(Complex z1, Complex z2)
73      {
74          return new Complex(z1.Re - z2.Re, z1.Im - z2.Im);
75      }
76
77      /// <summary>
78      /// Унарный минус используется для вычисления комплексного числа,
79      /// сопряженного к данному
80      /// </summary>
81      /// <param name="z1">Комплексное число</param>
82      /// <returns>Сопряженное комплексное число</returns>
83      public static Complex operator -(Complex z1)
84      {
85          return new Complex(z1.Re, -z1.Im);
86      }
87
88      /// <summary>
89      /// Бинарный оператор умножения двух комплексных чисел
90      /// </summary>
91      /// <param name="z1">Первый сомножитель (комплексное число)</param>
```

```
92      /// <param name="z2">Второй сомножитель (комплексное число)</param>
93      /// <returns>Произведение двух комплексных чисел</returns>
94      public static Complex operator *(Complex z1, Complex z2)
95      {
96          return new Complex(z1.Re * z2.Re - z1.Im * z2.Im,
97              z1.Im * z2.Re + z1.Re * z2.Im);
98      }
99
100     /// <summary>
101     /// Бинарный оператор деления двух комплексных чисел
102     /// </summary>
103     /// <param name="z1">Делимое (комплексное число)</param>
104     /// <param name="z2">Делитель (комплексное число)</param>
105     /// <returns>Частное от деления первого комплексного числа на второе.
106     /// Если в делителе находится 0, то результатом будет «не число»
107     /// </returns>
108     public static Complex operator /(Complex z1, Complex z2)
109     {
110         var zn = z2.Re * z2.Re + z2.Im * z2.Im;
111         return new Complex((z1.Re * z2.Re + z1.Im * z2.Im) / zn,
112             (z1.Im * z2.Re - z1.Re * z2.Im) / zn);
113     }
114
115     /// <summary>
116     /// Оператор сравнения двух комплексных чисел на равенство
117     /// Оператор сравнивает числа с точностью до
118     /// предпоследнего значащего бита
119     /// Например, числа 6.999999999999998 + i и 7 + i будут считаться равными
120     /// </summary>
121     /// <param name="z1">Комплексное число 1</param>
122     /// <param name="z2">Комплексное число 2</param>
123     /// <returns>true, если числа равны до предпоследнего значащего бита и
124     /// false в противном случае</returns>
125     public static bool operator ==(Complex z1, Complex z2)
126     {
127         // Вложенная функция для вычисления единицы наименьшей точности для числа
128         double Ulp(double value)
129         {
130             long bits = BitConverter.DoubleToInt64Bits(value);
131             double nextValue = BitConverter.Int64BitsToDouble(bits + 1);
132             return nextValue - value;
133         }
134
135         var reDelta = Ulp(z1.Re) + Ulp(z2.Re);
136         var imDelta = Ulp(z1.Im) + Ulp(z2.Im);
137         return z1.Re.AEq(z2.Re, reDelta) &&
138                z1.Im.AEq(z2.Im, imDelta);
139     }
140
141     /// <summary>
```

```
142         /// Оператор проверки чисел на неравенство
143         /// </summary>
144         /// <param name="z1">Первое комплексное число</param>
145         /// <param name="z2">Второе комплексное число</param>
146         /// <returns>true, если числа отличаются более чем на 1 наименьший разряд
147         /// false - в противном случае</returns>
148         public static bool operator !=(Complex z1, Complex z2)
149         {
150             return !(z1 == z2);
151         }
152
153         /// <summary>
154         /// Проверка двух чисел на идентичность по значению
155         /// </summary>
156         /// <param name="obj">Объект, с которым сравнивается
157         /// данное комплексное число</param>
158         /// <returns>true, если объект является комплексным числом и
159         /// его вещественная и мнимая части совпадают с вещественной и мнимой
160         /// частью данного комплексного числа</returns>
161         public override bool Equals(object? obj)
162         {
163             if (obj is Complex z)
164             {
165                 return z == this;
166             }
167
168             return false;
169         }
170
171         /// <summary>
172         /// Метод вычисления аргумента комплексного числа
173         /// </summary>
174         /// <returns>Аргумент комплексного числа</returns>
175         /// <returns>double.NaN, если комплексное число равно 0</returns>
176         public double Arg()
177         {
178             if (this == 0.0) return double.NaN;
179             return Math.Atan(Im / Re) +
180                 (Re > 0.0 ? 0.0 :
181                 Im > 0.0 ? Math.PI : -Math.PI);
182         }
183
184         /// <summary>
185         /// Метод вычисления модуля комплексного числа
186         /// </summary>
187         /// <returns>Модуль комплексного числа</returns>
188         public double Abs() => Math.Sqrt(Re * Re + Im * Im);
189
190         /// <summary>
191         /// Хэш-код, вычисленный по вещественной и мнимой части комплексного числа
```

```
192         /// </summary>
193         /// <returns>Хэш-код комплексного числа</returns>
194         public override int GetHashCode() => HashCode.Combine(Re, Im);
195
196         /// <summary>
197         /// Метод получения строкового представления комплексного числа
198         /// </summary>
199         /// <returns>Комплексное число в виде строки</returns>
200         public override string ToString()
201         {
202             var sb = new StringBuilder();
203             var delta = 1e-10;
204             var space = "";
205             if (!Re.AEq(0.0, delta) || Im.AEq(0.0, delta))
206             {
207                 sb.Append(Re);
208                 space = " ";
209             }
210             if (!Im.AEq(0.0, delta))
211             {
212                 sb.Append(space);
213                 sb.Append(Im > 0 ? '+' : '-');
214                 sb.Append(space);
215                 if (!Math.Abs(Im).AEq(1.0, delta))
216                     sb.Append(Math.Abs(Im));
217                 sb.Append('i');
218             }
219
220             return sb.ToString();
221         }
222
223         /// <summary>
224         /// Оператор неявного приведения типа вещественного числа в комплексное
225         /// </summary>
226         /// <param name="value">вещественное число,
227         /// преобразуемое в комплексное</param>
228         public static implicit operator Complex(double value)
229         {
230             return new Complex(value);
231         }
232
233         /// <summary>
234         /// Оператор явного приведения типа комплексного числа в вещественное.
235         /// В процессе преобразования отбрасывается мнимая часть.
236         /// </summary>
237         /// <param name="value">Преобразуемое комплексное число</param>
238         public static explicit operator double(Complex value)
239         {
240             return value.Re;
241         }
```

```
}

/// <summary>
/// Статический класс расширения вещественного типа
/// </summary>
public static class DoubleExtension
{
    /// <summary>
    /// Метод проверки вещественного числа на примерное равенство
    /// для сравнения с учетом возможной погрешности.
    /// При сравнении учитываются различие величин с заданной точностью
    /// </summary>
    /// <param name="first">Первое значение</param>
    /// <param name="second">Второе значение</param>
    /// <param name="delta">Точность, с которой сравниваются два значения</param>
    /// <returns>Результат сравнения чисел с заданной точностью на равенство</returns>
    public static bool AEq(this double first, double second, double delta)
    {
        return Math.Abs(first - second) < delta * 2;
    }
}
```

Протестировать созданный класс можно с помощью следующей программы.

Файл Program.cs.

```
using Operators;

var z0 = new Complex();
var z1 = new Complex(3, -5);
Complex z2 = 1.2;
var z3 = new Complex(0, -1);
var z4 = new Complex(-1, 1);
Console.WriteLine("z0 = {0}", z0);
Console.WriteLine("z1 = {0}", z1);
Console.WriteLine("z2 = {0}", z2);
Console.WriteLine("z3 = {0}", z3);
Console.WriteLine("z4 = {0}", z4);
Console.WriteLine("z1 + z3 = {0}", z1 + z3);
Console.WriteLine("z1 + 3 = {0}", z1 + 3);
Console.WriteLine("7 + z1 = {0}", 7 + z1);
Console.WriteLine("z4 - z1 = {0}", z4 - z1);
Console.WriteLine("z4 * z1 = {0}", z4 * z1);
Console.WriteLine("z1 / (-1 + 2i) = {0}",
    z1 / (z4 + new Complex(0, 1)));
Console.WriteLine("z1 / z0 = {0}", z1 / z0);
Console.WriteLine("z1 / 10 = {0}", z1 / 10);
Console.WriteLine("-z1 = {0}", -z1);
Console.WriteLine("+z1 = {0}", +z1);
var z5 = new Complex(1, 1);
Console.WriteLine($"z5={z5}, Arg={z5.Arg()}, Abs={z5.Abs()}");
```

```
26  var z6 = new Complex(6.999999999999998, 1);
27  var z7 = new Complex(7, 1);
28  var z8 = new Complex(6.99999999999998, 1);
29  Console.WriteLine($"z6 = {z6}, z7 = {z7}, z6==z7 = {z6 == z7}");
30  Console.WriteLine($"z8 = {z8}, z7 = {z7}, z8==z7 = {z8 == z7}");
31  var re9 = 1.34;
32  Complex z9 = 1.34;
33  Console.WriteLine("z9 = 1.34; z9 == 1.34? - {0}", z9 == re9);
34  Console.WriteLine("z9 = 1.34; z9.Equals(1.34)? - {0}", z9.Equals(re9));
35  var z10 = new Complex(1, -1);
36  var z11 = new Complex(-1, 1);
37  var z12 = new Complex(-1, 1);
38  Console.WriteLine("z10 = {3}, z11 = {4}, z12 = {5}, " +
39                    "HC(z10) = {0}, HC(z11) = {1}, HC(z12) = {2}",
40      z10.GetHashCode(),
41      z11.GetHashCode(),
42      z12.GetHashCode(),
43      z10, z11, z12);
```

На экран будет выведено.

```
z0 = 0
z1 = 3 - 5i
z2 = 1,2
z3 = -i
z4 = -1 + i
z1 + z3 = 3 - 6i
z1 + 3 = 6 - 5i
7 + z1 = 10 - 5i
z4 - z1 = -4 + 6i
z4 * z1 = 2 + 8i
z1 / (-1 + 2i) = -2,6 - 0,2i
z1 / z0 = не число - не числоi
z1 / 10 = 0,3 - 0,5i
-z1 = 3 + 5i
+z1 = 3 - 5i
z5=1 + i, Arg=0,7853981633974483, Abs=1,4142135623730951
z6 = 6,999999999999998 + i, z7 = 7 + i, z6==z7 = True
z8 = 6,99999999999998 + i, z7 = 7 + i, z8==z7 = False
z9 = 1.34; z9 == 1.34? - True
z9 = 1.34; z9.Equals(1.34)? - False
z10 = 1 - i, z11 = -1 + i, z12 = -1 + i, HC(z10) = 1049444288, HC(z11) = -
555965244, HC(z12) = -555965244
```

# 3. ФОРМИРОВАНИЕ ИЕРАРХИИ КЛАССОВ

## Наследование

Наследование является одним из основополагающих принципов объектно-ориентированного программирования. Благодаря наследованию один класс может перенять функциональность другого класса, расширив ее, либо

изменив, при этом программисту не потребуется в этом классе повторять код, который уже был написан ранее.

Класс, от которого другой унаследует поля методы и свойства будет называться базовым классом, родительским классом или суперклассом. Класс, который наследует что-то от другого будет называться производным классом, дочерним классом, наследником или потомком.

Описать класс-наследник можно указав после его имени через знак двоеточия название базового класса.

```
1  public class Parent
2  {
3      //…
4  }
5
6  public class Child : Parent
7  {
8      //…
9  }
```

В C# ВСЕ КЛАССЫ, ДАЖЕ ЕСЛИ ЭТО НЕ УКАЗАНО ЯВНО, НАСЛЕДУЮТ ОТ ОБЩЕГО ПРЕДКА – КЛАССА **`Object`**.

Отметим, что в языке C# при наследовании имеются некоторые ограничения.

1. Не поддерживается множественное наследование, то есть класс не может иметь более одного непосредственного родителя.
2. Уровень доступа к производному классу может быть либо таким же как у базового класса, либо более строгим. Например, если базовый класс имеет модификатор доступа **`internal`**, то производный класс может иметь уровень доступа не выше **`internal`**.
3. Если перед именем класса поставлено ключевое слово-модификатор **`sealed`**, то от этого класса нельзя создать производный класс.
4. Статический класс не может являться родителем для других классов.

**ОБРАТИТЕ ВНИМАНИЕ!** КОНСТРУКТОРЫ КЛАССА НЕ НАСЛЕДУЮТСЯ.

При создании объекта производного класса сначала всегда вызывается конструктор базового класса, который выполняется в первую очередь. Если это не указано явно, то будет вызван конструктор по умолчанию. Однако, если требуется вызывать иную реализацию, либо конструктор по умолчанию недоступен, нужная версия конструктора указывается явно, с использованием ключевого слова **`base`**, как показано ниже.

```
1  public class Parent
```

```
{
    public Parent(int i)
    {
        //…
    }
    //…
}

public class Child : Parent
{
    //…
    public Child(int i) : base(i) // явный вызов конструктора базового класса
    {
        //…
    }
}
```

Подобный же синтаксис можно использовать для вызова одной версии конструктора самого класса из другой, заменив слово **`base`** на **`this`**.

Следующий пример содержит более подробные описания двух классов **`A`** и **`B`**, один из которых является наследником другого (файл *ab.cs)*.

```
namespace Example3;

/// <summary>
/// Базовый класс
/// </summary>
public class A
{
    /// <summary>
    /// Закрытое поле
    /// </summary>
    private int _id;

    /// <summary>
    /// Защищенное свойство класса
    /// </summary>
    protected int Id => _id;

    /// <summary>
    /// Общедоступное строковое поле
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// Общедоступный метод класса для получения информации
    /// </summary>
    public string GetInfo()
    {
        return $"Класс A. Имя={Name}, Id={Id}.";
    }
```

```
30      /// <summary>
31      /// Конструктор класса создает экземпляр с заданным номером и именем
32      /// </summary>
33      /// <param name="id">Номер объекта</param>
34      /// <param name="name">Имя объекта</param>
35      public A(int id, string name)
36      {
37          _id = id;
38          Name = name;
39      }
40  }
41
42  /// <summary>
43  /// Класс-наследник (B наследует от A)
44  /// </summary>
45  public class B : A
46  {
47      public enum Type
48      {
49          Type1, Type2, Type3
50      }
51      /// <summary>
52      /// Свойство для хранения типа объекта
53      /// </summary>
54      public Type ObjectType { get; set; }
55
56      /// <summary>
57      /// Конструктор класса B вызывает конструктор класса A, с помощью которого
58      /// задает значения для унаследованных элементов (Id и Name),
59      /// а также устанавливает значение для своего собственного свойства ObjectType
60      /// </summary>
61      /// <param name="id">Идентификатор объекта</param>
62      /// <param name="name">Имя объекта</param>
63      /// <param name="type">Тип объекта</param>
64      public B(int id, string name, Type type) : base(id, name)
65      {
66          ObjectType = type;
67      }
68
69      /// <summary>
70      /// В классе необходимо описать свою реализацию метода GetInfo, чтобы
71      /// сообщать о том, что это наследник, а также выводить его тип.
72      /// </summary>
73      public string GetInfo()
74      {
75          return $"Класс B. Имя={Name}, Id={Id}, Тип={ObjectType}.";
76      }
77  }
```

Следующий код тестирует работу классов (файл main.cs).

```
1   var aobj1 = new A(1, "Первый");
```

```
2  var aobj2 = new A(2, "Второй");
3  Console.WriteLine(aobj1.GetInfo());
4  Console.WriteLine(aobj2.GetInfo());
5  var bobj1 = new B(1, "Первый", B.Type.Type1);
6  var bobj2 = new B(2, "Второй", B.Type.Type3);
7  Console.WriteLine(bobj1.GetInfo());
8  Console.WriteLine(bobj2.GetInfo());
```

При этом на экран будет выведена следующая информация.

```
Класс A. Имя=Первый, Id=1.
Класс A. Имя=Второй, Id=2.
Класс B. Имя=Первый, Id=1, Тип=Type1.
Класс B. Имя=Второй, Id=2, Тип=Type3.
```

### Преобразования типов

Объекты классов наследников всегда можно записать в переменную базового типа. При этом неявным образом выполняется так называемое восходящее преобразование.

```
9  A aobj3 = bobj2;
```

Однако обратное преобразование (нисходящее) неявным образом не выполняется.

```
10  B bobj3 = aobj1; // - ошибка. Неявное нисходящее преобразование невозможно
```

Это логично, поскольку класс-наследник в общем случае содержит больше информации, чем базовый класс. Для построения объекта базового класса по объекту наследника, в последнем информации будет достаточно, однако при выполнении обратной операции возникнет вопрос, чем заполнять поля и свойства наследника, которые в базовом классе не прописаны.

Тем не менее, в некоторых ситуациях такие преобразования возможны, однако требуют явного приведения типа.

Возможны два варианта преобразования:

- в стиле языка C++;
- с применением оператора **`as`**.

```
11  B bobj4 = (B)aobj1;
12  B? bobj5 = aobj1 as B;
```

В обоих случаях будет устранена синтаксическая ошибка, но все же в 11 строке возникнет исключительная ситуация, поскольку по-прежнему остается неясным, каким образом заполнить тип объекта (свойство **`ObjectType`**) **`bobj4`**. Если строку 11 закомментировать, то 12 строка отработает без сбоев, однако в **`bobj5`** попадет значение **`null`** по той же причине, что будет свидетельствовать о неуспешно выполненном преобразовании.

Следует отметить, что использование оператора **as** является предпочтительным, поскольку обработка исключительной ситуации, как правило, бывает более ресурсоёмкой операцией, чем проверка результата на равенство **null** с помощью условного оператора. Подробнее об обработке исключительных ситуаций написано в следующей главе.

Такие преобразования будут удачными только если преобразуемые объекты действительно являются представителями класса-наследника. Например, так может быть, если значение переменной базового класса получено посредством восходящего преобразования типа.

Третий вариант преобразования – использования оператора проверки принадлежности типа. Рассмотрим его на примере переменной, полученной путем восходящего преобразования.

```
13  A aobj4 = bobj2;
14  if (aobj4 is B bobj6)
15  {
16      Console.WriteLine(bobj6.ObjectType);
17  }
```

Здесь **aobj4** имеет тип **A**, однако в памяти по адресу, находящемуся в этой переменной, реально лежит объект класса **B**. Поэтому условие в 14 строке окажется истинным и результат преобразования будет записан в переменную **bobj6**, для которой в 16 строке будет выполнено требуемое действие.

В том случае, когда требуется перевести «настоящий» объект базового класса к дочернему, необходимо создать конструкторы нисходящего преобразования типа. Добавим с этой целью (для удобства использования) в класс **A** конструктор копирования.

```
    //...
32      public A(int id, string name)
33      {
34          _id = id;
35          Name = name;
36      }
37
38      /// <summary>
39      /// Конструктор копирования
40      /// </summary>
41      /// <param name="other">Объект класса, для которого требуется создать копию</param>
42      public A(A other)
43      {
44          _id = other.Id;
45          Name = other.Name;
46      }
47  }
48  // ...
```

И далее, в класс B – конструктор преобразования типа.

```
    // ...
71      public B(int id, string name, Type type) : base(id, name)
72      {
73          ObjectType = type;
74      }
75
76      /// <summary>
77      /// Конструктор нисходящего преобразования типа
78      /// </summary>
79      /// <param name="obj">Объект класса A</param>
80      public B(A obj) : base(obj)
81      {
82          ObjectType = Type.Type2;
83      }
84  // ...
```

Тогда в программе станет возможной операция по созданию наследника на основе экземпляра родительского класса.

```
18  B obj7 = new (aobj1);
```

### Виртуальные и переопределенные методы

Рассмотрим теперь следующую ситуацию. Пусть имеется функция для выполнения каких-либо операций с объектами классов **A** и **B**, а также объекты этих классов.

```
19  A saobj = new(10, "Объект для показа 1");
20  B sbobj = new(20, "Объект для показа 2", B.Type.Type1);
21  ShowInfo(saobj);
22  ShowInfo(sbobj);
23
24  void ShowInfo(A obj)
25  {
26      Console.WriteLine(obj.GetInfo());
27  }
```

Поскольку преобразование от типа наследника к типу предка всегда возможно, ошибки в 22 строке не возникнет.

Однако, на экран в этом случае будет выведено следующее.

```
Класс A. Имя=Объект для показа 1, Id=10.
Класс A. Имя=Объект для показа 2, Id=20.
```

Здесь оба объекта были показаны как объекты класса **A**, хотя на самом деле в 22 строке в функцию передается объект класса **B**.

Чтобы при вызове метода `GetInfo()` срабатывала реализация, принадлежащая именно тому классу, объект которого реально находится в памяти, компилятору необходимо дать инструкцию сформировать исполняемый

файл таким образом, чтобы вызываемый метод определялся в момент выполнения программы, а не на этапе компиляции. Для этого, необходимо:

1) в базовом классе пометить метод виртуальным с помощью ключевого слова **`virtual`**;
2) в производном классе пометить метод переопределенным с помощью ключевого слова **`override`**.

Тогда код программы примера будет выглядеть следующим образом.

```
    // …
23      public string GetInfo()
23      public virtual string GetInfo() // Объявление метода в классе A
        // …
89      public string GetInfo()
89      public override string GetInfo() // Объявление метода в классе B
        // …
```

Если после этих изменений запустить программу, на экране появится следующая информация.

```
Класс A. Имя=Объект для показа 1, Id=10.
Класс B. Имя=Объект для показа 2, Id=20, Тип=Type1.
```

---

**Обратите внимание!** Виртуальными и переопределенными могут быть не только методы, но и свойства. Для таких целей ключевые слова **`VIRTUAL`** и **`OVERRIDE`**, аналогично методам, ставятся перед типом свойства.

---

Например, для объявления виртуального свойства можно написать:

```
1   public virtual string Name
2   {
3       get => name;
4       set => name = value;
5   }
```

Тогда переопределенное свойство в классе-наследнике может выглядеть так:

```
1   public override string Name
2   {
3       get => $"Имя: {base.Name}";
4       set => base.Name = value;
5   }
```

### Абстрактные классы

Предположим, что программисту требуется реализовать набор классов для работы с различными геометрическими фигурами, например, квадратом, треугольником, кругом и т.д. Для каждой фигуры необходимо будет рассчитывать периметр (длину окружности) и площадь, а также хранить цвет фигуры. Поэтому в каждом из классов должны быть соответствующие свойства и методы. Очевидно, что в этом случае со всеми фигурами можно работать единообразно. Для этого нужно будет создать класс, объединяющий все общие компоненты, который будет являться базовым для всех фигур.

Однако в этом случае возникает вопрос, как именно реализовать алгоритм вычисления площади или периметра (длины окружности), ведь общей формулы для всех фигур может не быть. Именно в этом случае, свойства для получения площади и длины контура можно оставить нереализованными – абстрактными. Класс, который содержит хотя бы один абстрактный метод или свойство также называется ***абстрактным***. В этом случае перед нереализованным компонентом класса, а также перед самим классом, ставится ключевое слово `abstract`.

В следующем примере приведен код соответствующих классов.

Figure.cs – файл с абстрактным классом «фигура», содержащим общие поля, методы и свойства для всех фигур.

```
1   using System.Drawing;
2   using System.Text;
3
4   namespace AbstractModel;
5   public abstract class Figure
6   {
7       /// <summary>
8       /// Поле для хранения названия фигуры соответствующего вида
9       /// </summary>
10      private readonly string Name;
11
12      /// <summary>
13      /// Поле для хранения размеров фигуры
14      /// </summary>
15      protected double[] Sizes;
16
17      /// <summary>
18      /// Свойство для получения значений размеров фигуры
19      /// </summary>
20      public double[] FigureSizes
21      {
22          get
23          {
```

```
24              var cpy = new double[Sizes.Length];
25              Array.Copy(Sizes, cpy, Sizes.Length);
26              return cpy;
27          }
28      }
29
30      /// <summary>
31      /// Абстрактные классы могут содержать поля
32      /// </summary>
33      private Color _color;
34
35      /// <summary>
36      /// Реализованное свойство класса
37      /// </summary>
38      public Color FigureColor
39      {
40          get => _color;
41          set => _color = value;
42      }
43
44      /// <summary>
45      /// Площадь - это абстрактное (не реализованное) вычисляемое свойство
46      /// </summary>
47      public abstract double? Square { get; }
48
49      /// <summary>
50      /// Длина контура - это еще одно абстрактное свойство класса Фигура
51      /// </summary>
52      public abstract double? ContourLength { get; }
53
54      /// <summary>
55      /// Абстрактные классы могут содержать и конструкторы
56      /// </summary>
57      /// <param name="color">Цвет фигуры</param>
58      /// <param name="sizeCount">Количество величин, задающих размеры фигуры</param>
59      public Figure(string name, Color color, int sizeCount)
60      {
61          Name = name;
62          Sizes = new double[sizeCount];
63          FigureColor = color;
64      }
65
66      /// <summary>
67      /// Метод для получения строки с параметрами текущей фигуры
68      /// </summary>
69      /// <returns>Название, цвет и размеры геометрической фигуры в виде строки</returns>
70      public override string ToString()
71      {
72          StringBuilder sb = new StringBuilder(
73              $"{Name}. Цвет: {FigureColor.Name}.\nРазмеры: ");
```

```
74          foreach (var sz in FigureSizes)
75          {
76              sb.Append(sz);
77              sb.Append("; ");
78          }
79          return sb.ToString().TrimEnd(' ', ';');
80      }
81  }
```

Для класса **Figure** создано три наследника: круг (**Circle**), прямоугольник (**Rectangle**) и треугольник (**Triangle**).

Файл Circle.cs.

```
1   using System.Drawing;
2   namespace AbstractModel;
3   /// <summary>
4   /// Класс круг
5   /// </summary>
6   public class Circle : Figure
7   {
8       /// <summary>
9       /// Свойство для получения и задания радиуса.
10      /// При изменении значения, радиус можно установить не меньше 0.0
11      /// </summary>
12      public double Radius
13      {
14          get => Sizes[0];
15          set => Sizes[0] = Math.Max(0.0, value);
16      }
17
18      /// <summary>
19      /// Конструктор класса Круг
20      /// </summary>
21      /// <param name="color">Цвет круга</param>
22      /// <param name="radius">Радиус круга</param>
23      public Circle(Color color, double radius) :
24          base("Круг", color, 1)
25      {
26          Radius = radius;
27      }
28
29      /// <summary>
30      /// Вычисляемое переопределенное свойство для получения площади круга
31      /// </summary>
32      public override double? Square => Math.PI * Radius * Radius;
33
34      /// <summary>
35      /// Вычисляемое переопределенное свойство для получения длины окружности
36      /// </summary>
37      public override double? ContourLength => 2 * Math.PI * Radius;
38  }
```

Файл Rectangle.cs.

```
1   using System.Drawing;
2
3   namespace AbstractModel;
4   public class Rectangle : Figure
5   {
6       /// <summary>
7       /// Свойство для получения и контролируемого задания длины стороны A
8       /// </summary>
9       public double A
10      {
11          get => Sizes[0];
12          set => Sizes[0] = Math.Max(double.Epsilon, value);
13      }
14
15      /// <summary>
16      /// Свойство для получения и контролируемого задания длины стороны B
17      /// </summary>
18      public double B
19      {
20          get => Sizes[1];
21          set => Sizes[1] = Math.Max(double.Epsilon, value);
22      }
23
24      /// <summary>
25      /// Конструктор класса прямоугольник
26      /// </summary>
27      /// <param name="color">Цвет фигуры</param>
28      /// <param name="a">Длина стороны A</param>
29      /// <param name="b">Длина стороны B</param>
30      public Rectangle(Color color, double a, double b) :
31          base("Прямоугольник", color, 2)
32      {
33          A = a;
34          B = b;
35      }
36
37      /// <summary>
38      /// Переопределенное свойство для вычисления площади прямоугольника
39      /// </summary>
40      public override double? Square => A * B;
41
42      /// <summary>
43      /// Переопределенное свойство для вычисления периметра прямоугольника
44      /// </summary>
45      public override double? ContourLength => 2.0 * (A + B);
46  }
```

Файл Triangle.cs.

```
1   using System.Drawing;
2
3   namespace AbstractModel;
4
5   /// <summary>
6   /// Класс треугольник
7   /// </summary>
8   public class Triangle : Figure
9   {
10      /// <summary>
11      /// Длина стороны A
12      /// </summary>
13      public double A
14      {
15          get => Sizes [0];
16          set => Sizes[0] = Math.Max(double.Epsilon, value);
17      }
18
19      /// <summary>
20      /// Длина стороны B
21      /// </summary>
22      public double B
23      {
24          get => Sizes[1];
25          set => Sizes[1] = Math.Max(double.Epsilon, value);
26      }
27
28      /// <summary>
29      /// Длина стороны C
30      /// </summary>
31      public double C
32      {
33          get => Sizes[2];
34          set => Sizes[2] = Math.Max(double.Epsilon, value);
35      }
36
37      /// <summary>
38      /// Проверка того, что объект с указанными сторонами
39      /// может являться треугольником
40      /// </summary>
41      public Boolean IsTriangle => A < B + C &&
42                                   B < A + C &&
43                                   C < A + B;
44
45      /// <summary>
46      /// Конструктор треугольника предусматривает его создание по длинам трех сторон
47      /// и цвету
48      /// </summary>
49      /// <param name="color">Цвет фигуры</param>
```

```
50      /// <param name="a">Длина стороны A</param>
51      /// <param name="b">Длина стороны B</param>
52      /// <param name="c">Длина стороны C</param>
53      public Triangle(Color color, double a, double b, double c) :
54          base("Треугольник", color, 3)
55      {
56          A = a;
57          B = b;
58          C = c;
59      }
60
61      /// <summary>
62      /// Переопределенное свойство для нахождения площади треугольника
63      /// в случае, если он существует для указанных длин сторон
64      /// </summary>
65      public override double? Square
66      {
67          get
68          {
69              var np = ContourLength / 2;
70              if (np is {} p)
71                  return Math.Sqrt(p * (p - A) * (p - B) * (p - C));
72              return null;
73          }
74      }
75
76      /// <summary>
77      /// Переопределенное свойство для нахождения периметра треугольника,
78      /// если этот треугольник существует для указанных длин сторон
79      /// </summary>
80      public override double? ContourLength
81      {
82          get
83          {
84              if (IsTriangle)
85              {
86                  return A + B + C;
87              }
88              return null;
89          }
90      }
91
92      /// <summary>
93      /// Переопределяем метод ToString, поскольку метод базового класса
94      /// не может учесть всех особенностей фигуры "Треугольник".
95      /// </summary>
96      /// <returns> Строка с информацией о треугольнике</returns>
97      public override string ToString()
98      {
99          if (!IsTriangle)
```

```
            return "Фигура с такими сторонами не является треугольником";
        return base.ToString();
    }
}
```

Для тестирования созданных классов можно создать следующий код (Program.cs).

```
using System.Drawing;
using AbstractModel;

var c = new Circle(Color.Blue, 1);
var r = new AbstractModel.Rectangle(Color.FromArgb(10, 255, 0), 3, 5.5);
var t = new Triangle(Color.Aqua, 3, 4, 5);
var nonT = new Triangle(Color.Red, 1, 1, 10);
Show(c);
Show(r);
Show(t);
Show(nonT);

// Функция для отображения на экране информации о фигуре.
// Фигура передается через параметр общего типа Figure, но за счет наличия
// виртуальных методов, отображаются данные для конкретной фигуры.
void Show(Figure f)
{
    Console.WriteLine(f);
    if (f.ContourLength is {} ln) // Проверка, что периметр вычислен
        Console.WriteLine("Длина контура: {0}.", ln);
    if (f.Square is {} sq) // Проверка, что площадь не null (вычислена)
    Console.WriteLine("Площадь: {0}.", sq);
    Console.WriteLine();
}
```

В результате работы приложения на экране появится следующая информация.

Следует обратить внимание, что последний созданный объект (**nonT**) не является треугольником, поскольку его стороны не сомкнутся. В связи с этим, площадь и периметр фигуры не выводятся.

```
Круг. Цвет: Blue.
Размеры: 1
Длина контура: 6,283185307179586.
Площадь: 3,141592653589793.

Прямоугольник. Цвет: ff0aff00.
Размеры: 3; 5,5
Длина контура: 17.
Площадь: 16,5.

Треугольник. Цвет: Aqua.
```

```
Размеры: 3; 4; 5
Длина контура: 12.
Площадь: 6.

Фигура с такими сторонами не является треугольником
Длина контура: .
Площадь: .
```

### Интерфейсы

Интерфейс представляет собой ссылочный тип, который может определять некоторый функционал – набор методов и свойств, как правило без их реализации. При этом у программиста нет возможности создавать экземпляры интерфейсного типа. Заявленный в нём функционал должен быть реализован некоторым классом. При этом интерфейсы не могут содержать полей и конструкторов.

Сама необходимость в наличии интерфейсов как таковых в языке программирования C# связана с тем, что в этом языке, как и во многих других современных языках, запрещено множественное наследование. Любой класс или структура может наследовать только от одного класса-предка. Однако иногда требуется обеспечить получение потомком различных возможностей от нескольких предков. С этой целью классы могут реализовывать произвольное количество интерфейсов. Класс, реализующий интерфейс, обязан содержать те методы и свойства, которые заявлены, но не реализованы в интерфейсе, либо он будет считаться абстрактным.

Имена интерфейсов принято начинать с заглавной буквы «**I**» (от слова Interface). При этом зачастую их имена заканчиваются суффиксом –able, что говорит о том, что интерфейсы определяют некоторые способности (англ. «able» — уметь, быть способным сделать что-то).

Таким образом, интерфейсы могут определять следующие элементы:

- методы;
- свойства;
- индексаторы;
- события;
- статические поля и константы.

Ниже приведен пример описания интерфейса, содержащего все указанные виды элементов.

Пример использования интерфейсного класса приведен ниже.

Пусть в файле IConvertable.cs определен интерфейс.

```
1  namespace Convert;
2  /// <summary>
3  /// Интерфейс для классов, выполняющих различные преобразования
4  /// </summary>
5  public interface IConvertable
6  {
7      /// <summary>
8      /// Свойство для хранения обозначения первой величины
9      /// </summary>
10     string LabelFrom { get; }
11
12     /// <summary>
13     /// Свойство для хранения обозначения второй величины
14     /// </summary>
15     string LabelTo { get; }
16
17     /// <summary>
18     /// Метод прямого преобразования
19     /// </summary>
20     /// <param name="value">Величина в исходной шкале</param>
21     /// <returns>Величина в конечной шкале</returns>
22     double ConvertFore(double value);
23
24     /// <summary>
25     /// Метод обратного преобразования
26     /// </summary>
27     /// <param name="value">Величина в конечной шкале</param>
28     /// <returns>Величина в исходной шкале</returns>
29     double ConvertBack(double value);
30 }
```

Тогда в программе можно создать классы, его реализующие.

Класс-преобразователь температур (FahrenheitToCelsius.cs).

```
1  namespace Convert;
2  public class FahrenheitToCelsius : IConvertable
3  {
4      /// <summary>
5      /// Метка исходного значения - градусы Цельсия
6      /// </summary>
7      public string LabelFrom => "°F";
8
9      /// <summary>
10     /// Метка конечного значения - градусы Фаренгейта
11     /// </summary>
12     public string LabelTo => "°C";
13
14     /// <summary>
15     /// Выполнение прямого преобразования из градусов Фаренгейта
16     /// в градусы по Цельсию
17     /// </summary>
18     /// <param name="value">Температура в градусах по Фаренгейту</param>
```

```
19          /// <returns>Температура в градусах по Цельсию</returns>
20          public double ConvertFore(double value) => 5.0 / 9.0 * (value - 32);
21
22          /// <summary>
23          /// Выполнение обратного преобразования из градусов Цельсия
24          /// в градусы по Фаренгейту
25          /// </summary>
26          /// <param name="value">Температура в градусах по Цельсию</param>
27          /// <returns>Температура в градусах по Фаренгейту</returns>
28          public double ConvertBack(double value) => 9.0 / 5.0 * value + 32;
29      }
```

Для преобразования между температурными шкалами существует фиксированная формула перевода. Однако в некоторых случаях, например, при работе с валютами, для выполнения преобразований необходимо указывать актуальный курс преобразования. Для того, чтобы обеспечить в соответствующих классах наличие этого свойства, можно создать еще один интерфейс (**`IWithRate`**).

```
1   namespace Convert;
2
3   /// <summary>
4   /// Интерфейс, гарантирующий наличие свойства "Курс"
5   /// в классах, его реализующих
6   /// </summary>
7   internal interface IWithRate
8   {
9       double Rate { get; }
10  }
```

Для создания класса-преобразователя валют (**`CurrencyConverter`**) используем оба созданных интерфейса.

Класс-преобразователь валют (файл CurrencyConverter.cs)

```
1   namespace Convert;
2
3   /// <summary>
4   /// Класс преобразования между двумя различными валютами
5   /// </summary>
6   public class CurrencyConverter : IConvertable, IWithRate
7   {
8       /// <summary>
9       /// Свойство, хранящее курс преобразования
10      /// </summary>
11      public double Rate { get; set; }
12
13      /// <summary>
14      /// Символ первой валюты
15      /// </summary>
16      public string LabelFrom { get; init; }
17
```

```
18      /// <summary>
19      /// Символ второй валюты
20      /// </summary>
21      public string LabelTo { get; init; }
22
23      /// <summary>
24      /// Конструктор класса
25      /// </summary>
26      /// <param name="rate">Текущий курс валютной пары</param>
27      /// <param name="labelFrom">Символ исходной валюты</param>
28      /// <param name="labelTo">Символ конечной валюты</param>
29      public CurrencyConverter(
30          double rate,
31          string labelFrom,
32          string labelTo)
33      {
34          Rate = rate;
35          LabelFrom = labelFrom;
36          LabelTo = labelTo;
37      }
38
39      /// <summary>
40      /// Метод прямого преобразования между валютами по курсу
41      /// </summary>
42      /// <param name="value">Сумма в исходной валюте</param>
43      /// <returns>Сумма в конечной валюте</returns>
44      public double ConvertFore(double value) =>
45          value * Rate;
46
47      /// <summary>
48      /// Метод обратного преобразования между валютами по курсу
49      /// </summary>
50      /// <param name="value">Сумма в конечной валюте</param>
51      /// <returns>Сумма в исходной валюте</returns>
52      public double ConvertBack(double value) =>
53          value / Rate;
54  }
```

Пример использования созданных классов (Main.cs):

```
1   using Convert;
2
3   // Функция, принимающая в качестве параметров,
4   // преобразуемое значение и класс-преобразователь
5   void ShowResults(double valueToConvert, IConvertable converter)
6   {
7       Console.OutputEncoding = System.Text.Encoding.UTF8;
8       // Формат вывода будет содержать 2 знака после запятой для валютных преобразований
9       // и необходимое количество знаков после запятой для других вариантов.
10      var format =
```

```
11          converter is IWithRate ? "{0:F2}{1} = {2:F2}{3}"
12                                 : "{0:G4}{1} = {2:G4}{3}";
13      Console.WriteLine(
14          format,
15          valueToConvert,
16          converter.LabelFrom,
17          converter.ConvertFore(valueToConvert),
18          converter.LabelTo
19          );
20      Console.WriteLine(
21          format,
22          valueToConvert,
23          converter.LabelTo,
24          converter.ConvertBack(valueToConvert),
25          converter.LabelFrom
26      );
27      Console.WriteLine();
28  }
29
30  var yuanRubCnv = new CurrencyConverter(13.0, "¥", "₽");
31  var dollarRubCnv = new CurrencyConverter(99.90, "$", "₽");
32  ShowResults(10, yuanRubCnv);
33  ShowResults(10, dollarRubCnv);
34
35  var tempCnv = new FahrenheitToCelsius();
36  ShowResults(100, tempCnv);
```

В результате работы построенного приложения на экран будет выведено следующее.

```
10,00¥ = 130,00₽
10,00₽ = 0,77¥

10,00$ = 999,00₽
10,00₽ = 0,10$

100°F = 37,78°C
100°C = 212°F
```

## 4. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ ПО C#

### Обработка исключительных ситуаций

В некоторых случаях выполнение программы не может продолжаться в нормальной режиме по причине возникновения ошибок времени выполнения, появление которых трудно предугадать. Например, программист написал код, который начинает работу с файлом. Открывает файл с внешнего накопителя, проверяет успешность открытия и начинает процесс чтения

данных. Однако в этот момент пользователь отключает внешнее устройство. Поскольку это происходит в произвольный момент времени, проверить такую ситуацию с помощью условного оператора не представляется возможным. В этом случае выполнение программы может быть прервано с ошибкой. Программа «упадет».

Для обеспечения нормального функционирования приложений при подобных ситуациях, в языке C# предусмотрен механизм их обработки. Для этого действия, которые могут потенциально привести к сбоям во время выполнения программы, нужно поместить в защищенный блок.

В самом общем случае он имеет следующую конструкцию.

```
try
{
    // потенциально опасные действия
}
catch (SomeException1) // перехват исключений по их типу
{
    // действия, которые нужно совершить при возникновении
    // исключения типа SomeException1
}
catch (SomeException2 ex) // перехват исключений по типу
{
    // действия, которые нужно совершить при возникновении
    // исключения типа SomeException2
}
catch // перехват всех исключений, которые еще не были перехвачены
{
    // действия, которые нужно совершить при возникновении
    // других видов исключений
}
finally //действия, обязательные к выполнению в любом случае
{
    // действия, которые выполняются вне зависимости от того,
    // произошло исключение или нет.
}
```

Количество блоков **`catch`** не ограничивается. Можно отлавливать и по-разному реагировать на исключения самых разных видов. Тип, который указывается в круглых скобках за словом **`catch`** должен являться наследником общего для всех исключения класса – **`Exception`**. При этом исключения более общего типа должны располагаться ниже своих наследников, потому что иначе классы исключений-наследников никогда не сработают.

Также необходимо отметить, что любой из приведенных в примере блоков **`catch`** может отсутствовать. Необязательным является и блок **`finally`**. Однако после блока **`try`** все же должен следовать хотя бы один из нижестоящих блоков.

Алгоритм работы приведенной конструкции представлен на рис. 2.

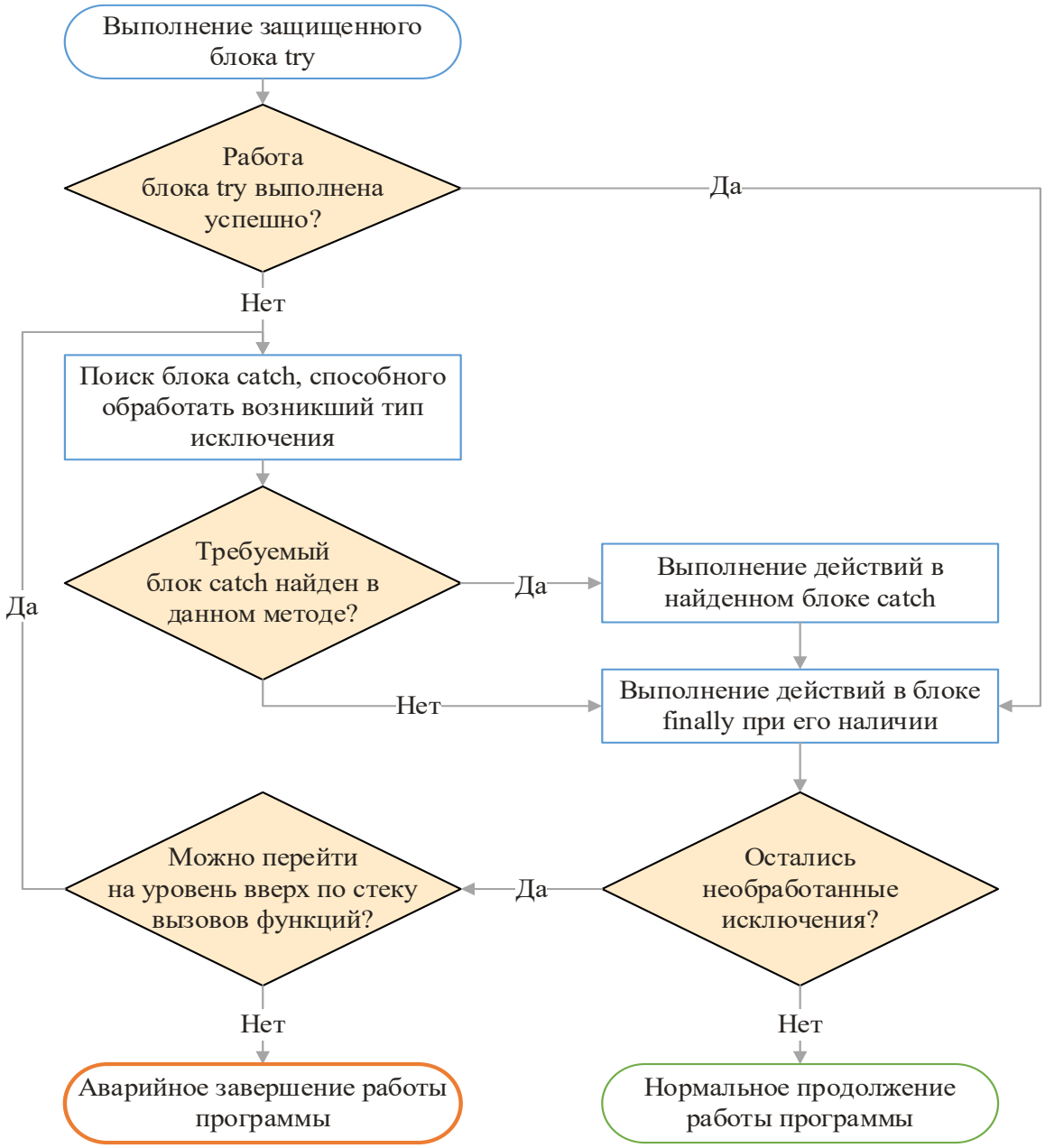


Рис. 2. Алгоритм работы защищенного блока

---

БЛОК **`finally`** НЕ МОЖЕТ СОДЕРЖАТЬ ОПЕРАТОРА ВЫХОДА ИЗ ФУНКЦИИ (**`return`**).

---

Основное назначение защищенного блока состоит в том, что он позволяет программисту разнести в пространстве кода возникновение и обработку ошибки времени выполнения. Если ошибка происходит в классе, который отвечает за вычисления, взаимодействие с пользователем в нем зачастую невозможно. В то же время класс может просигнализировать о возникновении ошибки другому классу, в задачу которого входит обеспечение интерфейса пользователя. В нем ошибку можно будет обработать и либо сообщить

пользователю о возникновении нештатной ситуации, либо выполнить иные действия по предотвращению краха всего приложения.

Для того, чтобы класс мог просигнализировать об ошибке используется ключевое слово **`throw`**, вслед за которым указывается объект типа-исключения (наследника класса **`Exception`**).

---

Существует также возможность указать ключевое слово **`throw`** без исключения-параметра, в случае, когда оно располагается в блоке **`catch`**. При этом будет повторно сгенерировано возникшее ранее исключение, и оно будет передано на обработку в вышестоящий блок **`catch`**.

---

Рассмотрим пример, содержащий методы обработки исключительных ситуаций.

Пусть имеется класс **`Product`**, содержащий некоторую информацию о продуктах, продаваемых в магазине, в том числе о их типе (еда, напитки, табак, алкоголь).

Файл Product.cs.

```
1   namespace TryCatchExample;
2
3   /// <summary>
4   /// Класс продукт
5   /// </summary>
6   public class Product
7   {
8       /// <summary>
9       /// Типы продуктов
10      /// </summary>
11      public enum Type
12      {
13          Еда, Напиток, Табак, Алкоголь
14      }
15
16      /// <summary>
17      /// Стоимость продукта
18      /// </summary>
19      private float _price;
20      /// <summary>
21      /// Тип продукта
22      /// </summary>
23      public Type ProductType { get; init; }
24      /// <summary>
25      /// Название продукта
26      /// </summary>
27      public string Name { get; set; }
28      /// <summary>
```

```
29          /// Свойство для получения стоимости продукта
30          /// </summary>
31          public float Price
32          {
33              get => _price;
34              init => _price = Math.Abs(value);
35          }
36          /// <summary>
37          /// Конструктор, создающий новый продукт по его параметрам
38          /// </summary>
39          /// <param name="name">Название</param>
40          /// <param name="productType">Тип продукта</param>
41          /// <param name="price">Стоимость продукта</param>
42          public Product(string name, Type productType, float price)
43          {
44              Name = name;
45              ProductType = productType;
46              Price = price;
47          }
48      }
```

Также создадим класс **Buyer**, содержащий информацию о покупателе товара (имя, возраст, имеющиеся в распоряжении средства, а также список купленных продуктов).

```
1   using System.Text;
2
3   namespace TryCatchExample;
4
5   /// <summary>
6   /// Класс покупателя
7   /// </summary>
8   public class Buyer
9   {
10      /// <summary>
11      /// Продуктовая корзина будет содержать купленные товары
12      /// </summary>
13      private List<Product> _cart = new ();
14
15      /// <summary>
16      /// Возраст покупателя
17      /// </summary>
18      private int _age;
19
20      /// <summary>
21      /// Количество денег у покупателя
22      /// </summary>
23      private float _money;
24
25      /// <summary>
26      /// Изначальное количество денег у покупателя
```

```
27          /// </summary>
28          private float _initialMoney;
29
30          /// <summary>
31          /// Имя покупателя
32          /// </summary>
33          public string Name { get; set; }
34
35          /// <summary>
36          /// Свойство для получения возраста покупателя
37          /// </summary>
38          public int Age
39          {
40              get => _age;
41              init => _age = Math.Abs(value);
42          }
43
44          /// <summary>
45          /// Свойство для получения и изменения количества денег у покупателя
46          /// </summary>
47          public float Money
48          {
49              get => _money;
50              private set => _money = Math.Abs(value);
51          }
52
53          /// <summary>
54          /// Конструктор, для создания нового покупателя
55          /// </summary>
56          /// <param name="name">Имя покупателя</param>
57          /// <param name="age">Возраст покупателя</param>
58          /// <param name="money">Первоначальная сумма в кошельке</param>
59          public Buyer(string name, int age, float money)
60          {
61              Name = name;
62              Age = age;
63              Money = money;
64              _initialMoney = Money;
65          }
66
67          /// <summary>
68          /// Покупка товара покупателем
69          /// </summary>
70          /// <param name="p">Покупаемый продукт</param>
71          /// <exception cref="IllegalAgeException">Исключение
72          /// происходит, если покупатель слишком молод для покупки
73          /// данного продукта</exception>
74          /// <exception cref="OutOfMoneyException">Исключение
75          /// выбрасывается если у покупателя осталось недостаточно средств
76          /// для покупки очередного продукта</exception>
```

```
77      public void Buy(Product p)
78      {
79          if (Age < 18 &&
80              (p.ProductType is Product.Type.Алкоголь ||
81               p.ProductType is Product.Type.Табак)
82             )
83              // Нельзя купить некоторые типы продуктов до 18 лет
84              throw new IllegalAgeException();
85          if (Money > p.Price)
86          {
87              _cart.Add(p);
88              Money -= p.Price;
89          }
90          else
91              // Невозможно купить товары, если недостаточно средств
92              throw new OutOfMoneyException(Name);
93      }
94
95      /// <summary>
96      /// Сводная информация о покупателе и его приобретениях
97      /// </summary>
98      /// <returns>Строка с информацией о покупателе</returns>
99      public override string ToString()
100     {
101         var sb = new StringBuilder()
102             .AppendLine($"Покупатель: {Name}. Возраст: {Age}.")
103             .AppendLine("Список покупок:");
104         foreach (var product in _cart)
105         {
106             sb.AppendLine($"{product.Name} ({product.ProductType})");
107         }
108
109         return sb.ToString();
110     }
111
112     /// <summary>
113     /// Метод, реализующий отмену покупки
114     /// </summary>
115     public void Reset()
116     {
117         Money = _initialMoney;
118         _cart.Clear();
119     }
120 }
```

Класс содержит метод осуществления покупки товара. Он может генерировать два вида исключения: **`IllegalAgeException`** (в случае, если покупатель пытается купить табак или алкоголь до достижения 18 лет) и **`OutOfMoneyException`** (при попытке купить продукт, который стоит больше, чем количество денег, которыми располагает покупатель).

Сами классы исключений описаны в файле Exceptions.cs.

```
1   namespace TryCatchExample;
2
3   /// <summary>
4   /// Класс исключения, сообщающего о слишком молодом возрасте покупателя
5   /// для покупки определенных товаров
6   /// </summary>
7   public class IllegalAgeException : Exception
8   {
9       public IllegalAgeException() :
10          base("Данный товар продается лицам старше 18 лет")
11      { }
12  }
13
14  /// <summary>
15  /// Класс исключения, сигнализирующего о недостаточности средств у покупателя
16  /// для покупки очередного товара из списка
17  /// </summary>
18  public class OutOfMoneyException : Exception
19  {
20      public OutOfMoneyException(string name) :
21          base($"У покупателя {name} закончились деньги")
22      { }
23  }
```

Процесс оплаты покупки реализован методом класса `CashRegister`, который приведен ниже. В нем также происходит частичная обработка возникающих ошибок.

Файл CashRegister.cs.

```
1   namespace TryCatchExample;
2
3   /// <summary>
4   /// Класс касса
5   /// </summary>
6   public class CashRegister
7   {
8       /// <summary>
9       /// Метод оплаты списка покупок
10      /// </summary>
11      /// <param name="buyer">Покупатель</param>
12      /// <param name="products">Список покупаемых продуктов</param>
13      public void Payment(Buyer buyer, List<Product> products)
14      {
15          try
16          {
17              Console.WriteLine($"У [{buyer.Name}] есть {buyer.Money:F2}р.");
18              // Перебираем все продукты
19              foreach (var product in products)
20              {
```

```
                Console.Write(
                    $"[{buyer.Name} ({buyer.Age})]" +
                    $" покупает {product.Name}" +
                    $" ({product.Price:F2}р.)... ");
                try
                {
                    buyer.Buy(product);
                    Console.WriteLine(
                        $"успешно. Остаток {buyer.Money:F2}р."
                        );
                }
                catch (IllegalAgeException ex)
                {
                    // Если человек не может купить по возрасту
                    Console.WriteLine("неудачно.");
                    Console.WriteLine(ex.Message);
                }
                catch
                {
                    // при любом другом исключении выводим сообщение о неудачности покупки
                    Console.WriteLine("неудачно.");
                    // и отправляем исключение на дальнейшую обработку
                    throw;
                }
            }
        }
        finally
        {
            // Обязательный вывод сообщения о завершении обслуживания на кассе
            Console.WriteLine(
                $"Процесс покупки для {buyer.Name} завершен.");
        }
    }
}
```

Внутренний блок `try` метода `Payment` позволяет организовать покупки с пропуском товаров, недоступных к продаже несовершеннолетним. Кроме того, этот блок выводит сообщение о неуспешности выполнения покупки по любой другой причине, однако в этом случае процесс оплаты товара прекращается, поскольку управление передается внешнему блоку `catch`. При этом применение блока `finally` гарантирует вывод сообщения об окончании процесса покупки при любом исходе.

Файл Program.cs содержит программный код для тестирования созданных классов. В нем создается тестовый список покупок, регистрируются покупатели, и для каждого из них предпринимается попытка купить товары из списка.

```
using TryCatchExample;
```

```
2   // Список имеющихся в магазине продуктов
3   List<Product> allProducts = new List<Product>()
4   {
5       new("Пельмени", Product.Type.Еда, 234.99f),
6       new("Сыр", Product.Type.Еда, 115.99f),
7       new("Сиг*****", Product.Type.Табак, 200.0f),
8       new("Конь**", Product.Type.Алкоголь, 599.99f),
9       new("Минеральная вода", Product.Type.Напиток, 13.99f)
10  };
11
12  // Информация о покупателях
13  var bInfo = new Dictionary<string, int>
14  {
15      {"Сергей", 24}, {"Ольга", 17}
16  };
17
18  // Список покупателей
19  var buyers = new List<Buyer>();
20
21  // Создание покупателей (экземпляров класса Buyer)
22  foreach (var (name, age) in bInfo)
23  {
24      buyers.Add(new Buyer(name, age, 500f));
25  }
26
27  // Создание кассы для оплаты покупок по списку
28  var sh = new CashRegister();
29  var productsToBuy = allProducts;
30  // Каждого покупателя отправляем в магазин за покупками
31  for (var i = 0; i < buyers.Count; i++)
32  {
33      try
34      {
35          // Визит в магазин не всегда заканчивается успешно, поэтому используем защищенный блок
36          sh.Payment(buyers[i], productsToBuy);
37      }
38      catch (OutOfMoneyException ex)
39      {
40          // Не хватило денег на покупку товаров из всего списка
41          Console.WriteLine(ex.Message);
42          // Попробуем купить только полезные товары, если это первая неудачная попытка
43          if (productsToBuy.Count == allProducts.Count)
44          {
45              // Покупатель возвращает уже купленные товары
46              buyers[i].Reset();
47              // Выбираем из списка продуктов только полезные товары
48              productsToBuy = allProducts.FindAll(m =>
49                  m.ProductType is not Product.Type.Алкоголь &&
50                  m.ProductType is not Product.Type.Табак);
51              // Этот же покупатель обслуживается еще раз
```

```
52              i--;
53              // Переход к следующей итерации цикла
54              continue;
55          }
56      }
57      finally
58      {
59          // Вывод разделителя производится даже после continue;
60          Console.WriteLine();
61      }
62      // Следующий покупатель будет пробовать сначала купить все товары
63      productsToBuy = allProducts;
64  }
65
66  // Итоговые покупки покупателей
67  foreach (var buyer in buyers)
68  {
69      Console.WriteLine(buyer);
70  }
```

В результате работы приложения на экран будет выведена следующая информация.

```
У [Сергей] есть 500,00р.
[Сергей (24)] покупает Пельмени (234,99р.)... успешно. Остаток 265,01р.
[Сергей (24)] покупает Сыр (115,99р.)... успешно. Остаток 149,02р.
[Сергей (24)] покупает Сиг***** (200,00р.)... неудачно
Процесс покупки для Сергей завершен.
У покупателя Сергей закончились деньги
Покупка полностью отменена.

У [Сергей] есть 500,00р.
[Сергей (24)] покупает Пельмени (234,99р.)... успешно. Остаток 265,01р.
[Сергей (24)] покупает Сыр (115,99р.)... успешно. Остаток 149,02р.
[Сергей (24)] покупает Минеральная вода (13,99р.)... успешно. Остаток 135,03р.
Процесс покупки для Сергей завершен.

У [Ольга] есть 500,00р.
[Ольга (17)] покупает Пельмени (234,99р.)... успешно. Остаток 265,01р.
[Ольга (17)] покупает Сыр (115,99р.)... успешно. Остаток 149,02р.
[Ольга (17)] покупает Сиг***** (200,00р.)... неудачно.
Данный товар продается лицам старше 18 лет
[Ольга (17)] покупает Конь** (599,99р.)... неудачно.
Данный товар продается лицам старше 18 лет
[Ольга (17)] покупает Минеральная вода (13,99р.)... успешно. Остаток 135,03р.
Процесс покупки для Ольга завершен.

Покупатель: Сергей. Возраст: 24.
Список покупок:
Пельмени (Еда)
Сыр (Еда)
```

```
Минеральная вода (Напиток)

Покупатель: Ольга. Возраст: 17.
Список покупок:
Пельмени (Еда)
Сыр (Еда)
Минеральная вода (Напиток)
```

## Обобщенные типы

В некоторых ситуациях, классы, которые мы пишем, должны быть предназначены для работы с различными типами данных. Например, в программе можно реализовать класс-пару, предназначенный для хранения каких-либо объектов. В этом случае, можно, конечно, указать, объекты каких именно типов требуется там хранить, однако при необходимости сохранения объекта любого другого типа, придётся писать новый класс. Это не выгодно.

Альтернативный способ – это использование универсального типа `object` для хранения объектов.

Тогда программа, содержащая такой класс могла бы выглядеть следующим образом.

```
1   var op = new OPair(45, "строка");
2   int x = (int)op.First;
3   int y = (int)op.Second; // здесь будет ошибка времени выполнения
4
5   Console.WriteLine(op); // (45, строка)
6
7   /// <summary>
8   /// Класс-пара для создания пар из двух объектов различных типов
9   /// </summary>
10  class OPair
11  {
12      /// <summary>
13      /// Первый член пары
14      /// </summary>
15      public object First { get; set; }
16
17      /// <summary>
18      /// Второй член пары
19      /// </summary>
20      public object Second { get; set; }
21
22      /// <summary>
23      /// Конструктор для создания пары
24      /// </summary>
25      /// <param name="first">Значение первого члена пары</param>
26      /// <param name="second">Значение второго члена пары</param>
```

```
27      public OPair(object first, object second)
28      {
29          First = first;
30          Second = second;
31      }
32
33      /// <summary>
34      /// Преобразование пары в строковый формат
35      /// </summary>
36      /// <returns>Строка с информацией о паре</returns>
37      public override string ToString()
38      {
39          return $"({First}, {Second})";
40      }
41  }
```

Отчасти это верно. Однако в этом случае можно столкнуться с несколькими проблемами.

Во-первых, при использовании типа **`object`**, будут выполняться преобразования типов (в случае, если используются типы-значения – так называемые упаковка при размещении значения в объекте **`object`** и распаковка при обратном преобразовании). Все это потребует от системы дополнительных ресурсов: памяти и времени. Если такие операции будут частыми, то это приведет к снижению производительности приложения.

Во-вторых, поскольку нам заранее может быть неизвестен тип реального содержимого пары, это может приводить к ошибкам при выполнении приложения, из-за некорректного преобразования типов, как это происходит в третьей строчке (зачеркнута) в последнем блоке кода. Второй элемент пары является строкой, однако компилятор не может этого проверить и не сообщает о синтаксической ошибке при попытке преобразования фактической строки в целое число.

Избавиться от этих проблем может помочь применение обобщённых типов.

Обобщенные типы данных в некотором смысле похожи на *шаблоны* в языке C++. Они также позволяют классу поддерживать работу с разными типами данных. Однако, обобщенный тип, в отличие от шаблона, осуществляет параметризацию не во время компиляции программы, а во время ее выполнения. Введение обобщенного типа выполняется добавлением к имени класса угловых скобок с одним или несколькими типами-параметрами внутри.

```
1  var p1 = new Pair<int, int>(3, 5);
2  var p2 = new Pair<string, double>("семь с половиной", 7.5);
3
4  Console.WriteLine(p1);  // (3, 5)
```

```
5   Console.WriteLine(p2); // (семь с половиной, 7.5)
6
7   /// <summary>
8   /// Класс-пара для создания пар из двух объектов различных типов
9   /// </summary>
10  /// <typeparam name="T1">Тип-параметр для первого члена пары</typeparam>
11  /// <typeparam name="T2">Тип-параметр для второго члена пары</typeparam>
12  class Pair<T1, T2>
13  {
14      /// <summary>
15      /// Первый член пары
16      /// </summary>
17      public T1 First { get; set; }
18
19      /// <summary>
20      /// Второй член пары
21      /// </summary>
22      public T2 Second { get; set; }
23
24      /// <summary>
25      /// Конструктор для создания пары
26      /// </summary>
27      /// <param name="first">Значение первого члена пары типа T1</param>
28      /// <param name="second">Значение второго члена пары типа T2</param>
29      public Pair(T1 first, T2 second)
30      {
31          First = first;
32          Second = second;
33      }
34
35      /// <summary>
36      /// Преобразование пары в строковый формат
37      /// </summary>
38      /// <returns>Строка с информацией о паре</returns>
39      public override string ToString()
40      {
41          return $"({First}, {Second})";
42      }
43  }
```

Таким образом, обобщённый класс `Pair` (пара) может хранить внутри себя два объекта произвольных, но заранее определенных типов. При его использовании не придется выполнять постоянные преобразования типов, а компилятор сможет заранее предупреждать о возможных проблемах.

При создании обобщенных классов у программиста имеется возможность задать ограничения на тип-параметр с помощью оператора `where`.

Пусть требуется создать класс-пару, допускающий сравнения его экземпляров. В этом случае класс пара может реализовать интерфейс `IComparable<Pair<T1, T2>>`. Однако для создания реализации такого класса, в

нем понадобится сравнивать значения элементов пары. С этой целью необходимо установить ограничения на типы `T1` и `T2`, чтобы они также были сравнимыми, то есть реализовывали интерфейс `IComparable<T>`. Проиллюстрируем это следующим примером.

```
var p1 = new Pair<int, double>(1, 3.5);
var p2 = new Pair<int, string>(1, "Это один");
var p3 = new Pair<int, string>(1, "Еще один");
var p4 = new Pair<int, string>(2, "Это два");
var p5 = new Pair<int, string>(0, null);
var p6 = new Pair<int, string>(0, "Это три");
// Console.WriteLine(p1.CompareTo(p2));  Ошибка: несравнимые типы
Console.WriteLine(p1.CompareTo(p1)); // 0
Console.WriteLine(p2.CompareTo(p3)); // 1
Console.WriteLine(p2.CompareTo(p4)); // -1
Console.WriteLine(p5.CompareTo(p6)); // -1

/// <summary>
/// Класс сравнимая пара
/// </summary>
/// <typeparam name="T1">Сравнимый тип-параметр первого значения</typeparam>
/// <typeparam name="T2">Сравнимый тип-параметр второго значения</typeparam>
class Pair<T1, T2> : IComparable<Pair<T1, T2>>
    where T1 : IComparable<T1>
    where T2 : IComparable<T2>
{
    T1? First { get; set; }
    T2? Second { get; set; }

    /// <summary>
    /// Создает пару
    /// </summary>
    /// <param name="first">Первое значение в паре</param>
    /// <param name="second">Второе значение в паре</param>
    public Pair(T1? first, T2? second)
    {
        First = first;
        Second = second;
    }

    /// <summary>
    /// Сравнивает две пары, если их элементы соответствуют по типам
    /// </summary>
    /// <param name="other">Пара для сравнения с текущей</param>
    /// <returns>-1, если первое значение в первой паре меньше
    /// первого значения второй пары, либо если первые значения в парах
    /// совпадают, а второе значение первой пары меньше второго значения второй пары;<br/>
    /// 0, если значения в парах идентичны;<br/>
    /// 1, если первое значение в первой паре больше первого значения во второй паре,
    /// либо если первые значения в парах совпадают, а второе значение в первой паре
```

```
46      /// больше второго значения во второй паре.</returns>
47      public int CompareTo(Pair<T1, T2>? other)
48      {
49          if (other == null) return 1;
50          var fstCmp = First?.CompareTo(other.First) ??
51                               (other.First == null ? 0 : -1);
52          if (fstCmp == 0)
53          {
54              return Second?.CompareTo(other.Second) ??
55                              (other.Second == null ? 0 : -1);
56          }
57          return fstCmp;
58      }
59
60      public override string ToString() => $"({First}, {Second})";
61  }
```

Отметим, что в ограничениях помимо указания наследования от некоторого класса или реализации интерфейса можно также указывать:

- `where T: class` – для указания того, что тип `T` должен являться классом;
- `where T: struct` – для указания на то, что тип `T` должен являться структурой;
- `where T: new()` – для обеспечения наличия в составе типа `T` публичного конструктора по умолчанию.

Также допускается указания нескольких ограничений для одного типа через запятую, например: `where T: class, IComparable<T>, new()`.

### Перечислимые типы

Перечислимым называется тип, который реализует интерфейс `IEnumerable`, и содержит метод `GetEnumerator()`, фактически возвращающий так называемый итератор. Использование этого метода в классе позволяет использовать объект этого класса в цикле `foreach`.

В следующем примере будет написан класс `Odd`, который позволит перебирать в цикле `foreach` нечетные числа в заданном диапазоне.

Файл Odd.cs.

```
1   using System.Collections;
2
3   namespace Enumerable;
4
5   /// <summary>
6   /// Перечислимый класс, позволяющий перебирать нечетные числа в заданном диапазоне
7   /// </summary>
8   public class Odd : IEnumerable
```

```
{
    /// <summary>
    /// Итератор, позволяющий переходить по нечетным числам в заданном диапазоне
    /// </summary>
    private IEnumerator _oddEnumarator;

    /// <summary>
    /// Конструктор для создания перечислителя нечетных значений в заданном диапазоне
    /// </summary>
    /// <param name="min">Наименьшее перечисляемое нечетное число</param>
    /// <param name="max">Наибольшее перечисляемое нечетное число</param>
    public Odd(int min, int max)
    {
        _oddEnumarator = new OddEnumerator(min, max);
    }

    /// <summary>
    /// Метод возвращает перечислитель нечетных чисел из диапазона
    /// </summary>
    /// <returns>Перечислитель</returns>
    public IEnumerator GetEnumerator() => _oddEnumarator;
}

class OddEnumerator : IEnumerator
{
    private int _current;
    private int _min;
    private int _max;

    public OddEnumerator(int min, int max)
    {
        // Определяем, какая из границ больше (меньше) и, при необходимости,
        // меняем значения местами. При выполнении операции используются
        // кортежи.
        (_min, _max) = min < max ? (min, max) : (max, min) ;
        Reset();
    }

    /// <summary>
    /// Переход к следующему перечисляемому значению
    /// </summary>
    /// <returns>true, если удалось перейти на следующее значение,
    /// false, если произошел выход за пределы допустимого диапазона</returns>
    public bool MoveNext()
    {
        _current += 2;
        return _current <= _max;
    }

    /// <summary>
```

```
59          /// Сброс перечислителя в исходное состояние
60          /// </summary>
61          public void Reset()
62          {
63              _current = _min - Math.Abs(_min % 2) - 1;
64          }
65
66          /// <summary>
67          /// Текущий перечисляемый элемент
68          /// </summary>
69          public object Current => _current;
70      }
```

Для реализации метода **`GetEnumerator()`** в данном примере был создан дополнительный класс **`OddEnumerator`**, реализующий в свою очередь интерфейс **`IEnumerator`**, в котором заявлены два метода – **`MoveNext()`** для перехода на следующий элемент и **`Reset()`** для перевода объекта в начальное состояние, а также одно свойство **`Current`** – для получения значения текущего элемента.

Протестировать работу коллекции можно на следующем примере (Program.cs).

```
1   using Enumerable;
2
3   var odd1 = new Odd(-3, 10);
4   foreach (int value in odd1)
5   {
6       Console.Write("{0} ", value);
7   }
8   Console.WriteLine();
```

На экране отобразятся нечетные числа от -3 до 9.

Тем не менее, данная реализация не является полноценной. Её недостатки проиллюстрирует следующий блок кода.

```
9
10  var odd2 = new Odd(2, 21);
11  foreach (int value in odd2)
12  {
13      Console.Write("{0} ", value);
14      if (value > 10) break;
15  }
16  Console.WriteLine();
17  foreach (int value in odd2)
18  {
19      Console.Write("{0} ", value);
20  }
21  Console.WriteLine();
```

Здесь в двух циклах используется один и тот же объект `odd2`. При этом первый цикл прерывается досрочно.

На экран будет выведено.

```
-3 -1 1 3 5 7 9
3 5 7 9 11
13 15 17 19 21
```

В итоге второй цикл продолжил перебор значений, вместо того чтобы начать работу с начала, как это, очевидно, ожидалось.

Чтобы скорректировать это поведение, необходимо, чтобы класс `OddEnumerator` также реализовал интерфейс `IDisposable`, а вместе с ним метод `Dispose()`.

В коде примера (Odd.cs) произойдут следующие изменения.

```
    // ...
32  class OddEnumerator : IEnumerator, IDisposable
        // ...
70
71      /// <summary>
72      /// Необходимо реализовать для возможности возобновления перечисления
73      /// в новых циклах foreach
74      /// </summary>
75      public void Dispose()
76      {
77          Reset();
78      }
79  }
```

После указанных изменений работа всех циклов будет соответствовать ожиданиям.

```
-3 -1 1 3 5 7 9
3 5 7 9 11
3 5 7 9 11 13 15 17 19 21
```

Полученная конструкция является полноценной, но довольно многословной. Для создания перечисления есть смысл воспользоваться оператором `yield return`, который будет генерировать перечислитель самостоятельно. В этом случае можно будет обойтись без явного создания вспомогательного класса и реализации интерфейса `IEnumerator`.

В следующем примере показан вариант создания перечислимого объекта для четных чисел. Его суть полностью аналогична предыдущему примеру, однако итератор в нем реализован на безе применения оператора `yield return`. Файл Even.cs приведен ниже.

```
1   using System.Collections;
2
3   namespace Meth_IEnumerable;
4
```

```
5   /// <summary>
6   /// Перечислимый класс, позволяющий перебирать нечетные числа в заданном диапазоне
7   /// </summary>
8   public class Even : IEnumerable
9   {
10      private int _min;
11      private int _max;
12
13      /// <summary>
14      /// Конструктор для создания перечислителя четных значений в заданном диапазоне
15      /// </summary>
16      /// <param name="min">Наименьшее перечисляемое четное число</param>
17      /// <param name="max">Наибольшее перечисляемое четное число</param>
18      public Even(int min, int max)
19      {
20          (_min, _max) = min < max ? (min, max) : (max, min);
21      }
22
23      /// <summary>
24      /// Метод получения перечислителя четных элементов
25      /// </summary>
26      /// <returns>Перечислитель четных элементов, сгенерированный
27      /// оператором yield return </returns>
28      public IEnumerator GetEnumerator()
29      {
30          for (int i = _min + (_min % 2 == 0 ? 0 : 1); i <= _max; i += 2)
31          {
32              yield return i;
33          }
34      }
35  }
```

Отличие оператора `yield return` от стандартного `return` состоит в том, что он не только возвращает указанное после него значение и выполняет выход из метода, но также осуществляет сохранение состояния выполнения метода, в котором он находится. При этом после повторного обращения к методу, восстанавливается сохраненное состояние и выполнение метода начинается не заново, а продолжается с той точки, где произошла остановка в прошлый раз.

---

Помимо оператора `yield return`, существует еще оператор `yield break`, который сообщает об окончании элементов в последовательности.

---

### Коллекции

Коллекции удобно использовать для решения многих задач программирования. Основным назначением коллекций является стандартизация обработки групп объектов в программе. Все коллекции, существующие в .NET,

разработаны на основе четко определенных интерфейсов. Как правило, классов стандартной библиотеки, реализующих эти интерфейсы, вполне достаточно для подавляющего большинства задач. Тем не менее, у программиста имеется возможность реализовать и собственную коллекцию.

В среде .NET поддерживаются несколько разновидностей коллекций. Рассмотрим наиболее употребимые их виды.

Самыми часто встречающимися в приложениях коллекциями являются *обобщённые коллекции*. Они позволяют организовать работу с большинством структур данных, используемых в прикладных задачах, таких как списки, стеки, очереди, словари и множества. Обобщенные коллекции являются строго типизированными.

Для работы с такими коллекциями необходимо использовать библиотеку `System.Collections.Generic`.

*Необобщенные коллекции* реализуют схожий функционал, однако хранят объекты типа `object`. Соответственно, использование таких коллекций менее безопасно, а работа с ними производится медленнее из-за операций преобразования типа, упаковки и распаковки (о чём говорилось в разделе об обобщенных типах данных). Соответственно такие виды коллекций имеет смысл использовать только в случаях, когда в них могут одновременно храниться объекты различных типов, что, пожалуй, является достаточно нестандартной ситуацией.

В таблице I-4 перечислены классы обобщенных коллекций и дается их описание. Также в ней приводятся названия аналогичных классов необобщенных коллекций (библиотека `System.Collections`).

**Таблица I-4**

**Основные обобщенные классы коллекций и их необобщенные аналоги**

| Обобщенные классы | Описание | Необобщенные классы |
|---|---|---|
| *Коллекции для работы со списками* | | |
| `List<T>` | Определяет массив переменной длины, т.е. такой массив, который может при необходимости изменять свой размер. | `ArrayList` |
| `LinkedList<T>` | Класс представляет собой двунаправленный список, задаваемый первым и последним элементами, в котором каждый элемент ссылается на следующий и | |

| Обобщенные классы | Описание | Необобщенные классы |
|---|---|---|
| | предыдущий. Преимущества такой коллекции состоят в том, что добавление и удаление элементов «в середину» происходят очень быстро, поскольку отсутствует необходимость сдвигать ранее существующие элементы. Однако имеются и недостатки. Так, доступ к элементам может быть осуществлен исключительно последовательно, начиная с первого или с последнего. | |
| *Коллекции пар ключ-значение* | | |
| `Dictionary<TKey, TValue>` | Сохраняет пары "ключ-значение", которые собраны в хэш-таблицу, где хэш-значение используется в качестве ключа. Класс `Dictionary` может использоваться в тех случаях, когда необходимо сохранить не только сами элементы, но и ключи для доступа к ним, которые, при этом, не являются подряд идущими натуральными числами. | `Hashtable` |
| `SortedList<TKey, TValue>` | Эти классы, как (`Dictionary<TKey, TValue>`) поддерживают коллекцию пар «ключ/значение», но используют для внутреннего хранения данных сам ключ, а не хэш-значения. Таким образом, в данном случае мы получаем набор элементов, отсортированных по значению ключа, а не по хэш-значению. Элементы коллекции могут быть доступны как по ключу, так и по *индексу*. | `SortedList` |
| `SortedDictionary<TKey, TValue>` | Классы `SortedDictionary<TKey, TValue>` и `SortedList<TKey, TValue>` имеют схожую функциональность, но различное внутреннее устройство:<br>• `SortedList` использует меньше памяти, чем `SortedDictionary`; | |

| Обобщенные классы | Описание | Необобщенные классы |
|---|---|---|
| | • **SortedDictionary** быстрее вставляет и удаляет элементы;<br>• **SortedList** позволяет обращаться к элементам не только по ключам, но и по индексам.<br>Заполненный класс типа **SortedList** работает быстрее, если при этом не требуется изменение емкости. | |
| *Коллекции для работы со стеком* | | |
| **Stack<T>** | Классы для работы со стеком. Стек позволяет организовать работу со списком по принципу «последним пришел – первым обслужен».<br>Классы реализуют стек с помощью таких методов, как **Push** и **Pop**, для помещения элементов на стек и удаления их с вершины стека. Метод **Peek** возвращает верхний элемент, оставляя его при этом в стеке. | **Stack** |
| *Коллекции для работы с очередями* | | |
| **Queue<T>** | Класс для работы с очередью, которая позволяет организовать работу по принципу «первым пришел – первым обслужен».<br>Для добавления и удаления объектов из очереди используются методы **Enqueue** и **Dequeue** соответственно. Метод **Peek** возвращает элемент с головы очереди, оставляя его при этом в коллекции. | **Queue** |
| **PriorityQueue <TElem, TPriority>** | Коллекция элементов, которые добавляются в очередь с приоритетом. Извлекаются элементы в соответствии с их приоритетом: от меньшего к большему, а в рамках одного и того же приоритета в порядке их попадания в очередь. Таким образом, **PriorityQueue** – это расширенная версия стандартной очереди. | |

| Обобщенные классы | Описание | Необобщенные классы |
|---|---|---|
| *Коллекции для работы со множеством неповторяющихся элементов* | | |
| `HashSet<T>` | Класс представлен только в обобщенном формате. Используются для хранения неповторяющихся элементов. | |
| `SortedSet<T>` | Класс представлен только в обобщенном формате. При этом в отличие от предыдущего типа коллекции, элементы здесь автоматически упорядочиваются. | |

*Параллельные коллекции,* определенные в пространстве имен **`System.Collections.Concurrent`**, поддерживают доступ с использованием нескольких потоков.

Еще одним полезным типом коллекции является обобщенная коллекция, определенная в пространстве имен **`System.Collections.ObjectModel`** и представленная классом **`ObservableCollection<T>`**. Данная коллекция похожа на список, однако помимо базовой функциональности позволяет уведомлять внешние объекты об изменении состава своих элементов.

Создание собственных коллекций возможно благодаря реализации одного из интерфейсов: **`IReadOnlyCollection`** (**`IReadOnlyCollection<T>`**) для создания коллекций, доступных только для чтения (их элементы могут быть фиксированными и не изменяются), либо **`ICollection`** (**`ICollection<T>`**) – интерфейса для создания полноценной коллекции.

Интерфейс коллекции только для чтения отличается от **`IEnumerable`** только наличием дополнительного свойства **`Count`**, предоставляющего доступ к информации о количестве элементов в коллекции.

Интерфейс полноценной коллекции (**`ICollection<T>`**) помимо уже упомянутых методов содержит также:

- метод **`Add(T item)`** для добавления элемента в коллекцию;
- метод **`Clear()`** для удаления всех элементов из коллекции;
- метод **`Contains(T item)`** для проверки наличия элемента в коллекции;
- метод **`CopyTo(T[] array, int arrayIndex)`** для копирования всех элементов коллекции в указанный массив, начиная с некоторого номера **`arrayIndex`**;
- метод **`Remove(T item)`** для удаления элемента из коллекции.

Также интерфейс добавляет свойство `bool IsReadOnly`, которое возвращает булевское значение, сообщающее о возможности или невозможности редактирования коллекции.

В необобщенном варианте интерфейс `ICollection` содержит немного иной набор методов и свойств:

- метод `GetEnumerator()`;
- метод `CopyTo(Array array, int index)`;
- свойство `int Count`;
- свойство `bool IsSynchronized`, сообщающее о том, является ли доступ к коллекции потокобезопасным;
- свойство `object SyncRoot`, содержащее объект для выполнения синхронизации доступа к коллекции.

## 5. ДЕЛЕГАТЫ. СОБЫТИЯ. ЛЯМБДА ВЫРАЖЕНИЯ

### Делегаты

Делегат – это особый тип данных, позволяющий хранить в переменных этого типа ссылки (указатели) на методы.

Для объявления делегата используется ключевое слово `delegate`, после которого указывается тип возвращаемого значения, название и список параметров в круглых скобках – всё, как при объявлении метода.

Пример объявления делегата приведен ниже.

```
1  public delegate double Fun(double x);
```

Для использования делегата понадобится переменная этого типа. После ее объявления, ей можно будет присвоить ссылку на любую конкретную функцию, которая будет обладать соответствующим типом возвращаемого значения и параметрами (в данном случае – принимать один параметр типа `double` и возвращать вещественное значение).

К такой переменной можно обратиться как к функции (осуществить вызов делегата).

В следующем примере описан метод форматированного вывода на экран строки, содержащей имя некоторой функции, значение ее параметра и получаемое в результате её вычисления значение, при том, что сама функция становится известна только при использовании метода печати, путем передачи ему функционального параметра типа делегата.

```
1  var x = Math.PI / 2.0;
2
3  Print(Math.Cos, x);
4  Print(Math.Sin, x);
```

```
5   Print(MyFuncs.TwoXTimesSin, x);
6
7   void Print(Fun f, double x)
8   {
9       Console.Write($"{f.Method.Name}({x}) = ");
10      var result = f(x);
11      Console.WriteLine(Math.Round(result, 15));
12  }
13
14  public delegate double Fun(double x);
15  class MyFuncs
16  {
17      public static double TwoXTimesSin(double x)
18      {
19          return 2 * x * Math.Sin(x);
20      }
21  }
```

Здесь, например, в третьей строке в метод **Print** передается функция **Math.Cos** и параметр **x** = $\pi/2$. Соответственно на печать будет выведено:

```
Cos(1,5707963267948966) = 0
```

Сначала метод **Print** получает информацию об имени переданного метода (строка 9) и значении параметра **x**. Затем, в строке 10, вызывает переданный делегат и получает результат вычислений, который затем выводится в строке 11.

Для двух следующих вызовов, на экране будут получены, соответственно, следующие строки.

```
Sin(1,5707963267948966) = 1
TwoXTimesSin(1,5707963267948966) = 3,141592653589793
```

Одному делегату можно присвоить сразу несколько методов, добавляя их поочередно посредством оператора «**+=**». Однако следует иметь в виду, что если делегат возвращает значение, то будет возвращен результат работы только последнего добавленного в переменную делегата. Это утверждение иллюстрирует следующий пример.

```
1   // Объявляем переменную типа делегата
2   MyDelegate? f = null;
3   // Добавляем в переменную сразу несколько ссылок на методы
4   f += F1;
5   f += F2;
6   f += F3;
7   // Производим вызов делегата
8   Console.WriteLine("Результат работы: {0}", f());
9
10  int F1()
11  {
12      Console.WriteLine("Работа функции F1");
```

```
13      return 1;
14  }
15
16  int F2()
17  {
18      Console.WriteLine("Работа функции F2");
19      return 2;
20  }
21
22  int F3()
23  {
24      Console.WriteLine("Работа функции F3");
25      return 3;
26  }
27
28  public delegate int MyDelegate();
```

В этом случае на экран будет выведено следующее сообщение.

```
Работа функции F1
Работа функции F2
Работа функции F3
Результат работы: 3
```

Здесь видно, что отработали все три функции (`F1`, `F2` и `F3`), а в качестве результата используется результат последней добавленной в переменную `f` функции – `F3`.

### Анонимные методы и лямбда-выражения

При работе с делегатами не обязательно создавать полноценные функции, соответствующие им по типу и параметрам. Достаточно создать анонимную функцию (функцию без имени) с помощью ключевого слова `delegate`. В следующем примере метод `Print` получает в качестве параметра анонимную функцию.

```
1   Print(delegate (double x)
2   {
3       return x * x;
4   }, 3);
5
6   void Print(Fun f, double x)
7   {
8       Console.Write($"{f.Method.Name}({x}) = ");
9       var result = f(x);
10      Console.WriteLine(Math.Round(result, 15));
11  }
12
13  public delegate double Fun(double x);
```

При этом на экране тогда будет отображаться примерно следующее.

```
<<Main>$>b__0_0(3) = 9
```

Анонимный метод также можно записать более кратко с использованием синтаксиса так называемых лямбда-выражений.

Для вышеприведенного случая запись может быть трансформирована в следующую.

```
1   Print((x) => x * x, x);
```

Ключевую роль в этой записи играет лямбда-оператор (**=>**). Слева от него в круглых скобках располагается список параметров лямбда-функции. Справа – выполняемое выражение или набор действий. Если лямбда состоит из нескольких выражений, их можно разместить в фигурных скобках.

```
1   Print((x) => { var k = x * x; return 2 + k; }, x);
```

Лямбда-выражения являются представителями делегата и не могут существовать без объявления соответствующего типа. В приведенных примерах тип параметра **x** лямбда-выражения определяется типом делегата **Fun**, принимаемого методом **Print**. Именно благодаря этому компилятор понимает, сколько параметров должна принимать лямбда, и что должна вернуть в качестве результата.

Для удобства в .NET существует ряд стандартных делегатов, которые могут быть использованы в некоторых типичных ситуациях. К ним можно отнести такие делегаты как **Action**, **Predicate** и **Func**.

Делегат **Action** представляет собой некоторое действие, которое ничего не возвращает. При этом объекты делегата **Action** могут принимать от 0 до 16 параметров. Для этого тип **Action** имеет перегруженные обобщенные варианты:

- **void Action();**
- **void Action<in T1>(T1 arg1)**;
- …
- **void Action<in T1, …, in T16>(T1 arg1, …, T16 arg16)**.

Здесь T1, … T16 – произвольные типы возможных параметров.

Так, если создать переменную типа **Action<int, float>**, то в нее можно будет положить, например, такую функцию как **void Fun(int a, float b)**.

Примером действия с одним параметром может служить лямбда, выводящая на экран переданное ей числовое значение.

```
1   Action<int> Act = (a) => Console.WriteLine($"a = {a}");
```

На делегат **Action** похож и делегат **Func**, который отличается от него лишь наличием возвращаемого значения произвольного типа. Соответственно для Func тоже существует 17 перегруженных вариантов:

- **TResult Func<out TResult>()**;
- **TResult Func<out TResult, in T1>(T1 arg1)**;
- …
- **TResult Func<out TResult, in T1, …, in T16>(T1 arg1, …, T16 arg16)**.

Например, рассмотренному типу будет соответствовать лямбда, определяющая максимальное из двух целых чисел:

```
1  Func<int, int, int> Max = (a, b) => (a >= b) ? a : b;
```

Отличительной особенностью стандартного делегата **Predicate** является то, что он может принимать один параметр произвольного типа, а возвращать значение типа **bool**.

```
1  Predicate<double> IsInt = (x) => x - (int)x <= 1e-15;
2
3  Console.WriteLine(IsInt(3.14));
4  Console.WriteLine(IsInt(3.0));
```

В приведенном примере вводится лямбда-предикат для проверки того, является ли переданное вещественное по типу значение фактически целым числом.

```
False
True
```

### Замыкания

Замыкание представляет собой специальный объект, создаваемый для некоторой функции, который хранит:

- вложенную функцию или лямбду;
- внешнюю функцию, а также ее параметры и локальные переменные – ее лексическое окружение.

Рассмотрим пример.

```
1  OutFun();
2  void OutFun()
3  {
4      const int count = 10;
5      int i;
6      void InnFun()
7      {
8          Console.WriteLine($"i = {i}.");
```

```
9       }
10
11      for (i = 0; i < count; i++)
12      {
13          InnFun();
14      }
15  }
```

В приведенном фрагменте кода внешней является функция **OutFun**. В ее лексическом окружении находятся константа **count** и локальная переменная **i**. Она содержит внутреннюю функцию **InnFun**. Благодаря наличию замыкания – объекта, создаваемого в памяти для **InnFun**, у последней имеется доступ к переменной **i** и она может выводить ее значение на экран (строка 8).

В результате работы кода будет выведена следующая последовательность строк.

```
i = 0.
i = 1.
i = 2.
i = 3.
i = 4.
i = 5.
i = 6.
i = 7.
i = 8.
i = 9.
```

Однако при использовании замыканий нужно быть внимательным.

Например, если переделать код следующим образом.

```
1   OutFun();
2   void OutFun()
3   {
4       const int count = 10;
5       Action[] innFuns  = new Action[count];
6       for (int i = 0; i < count; i++)
7       {
8           innFuns[i] = () => { Console.WriteLine($"i = {i}."); };
9       }
10      for (int j = 0; j < count; j++)
11      {
12          innFuns[j]();
13      }
14  }
```

На экране окажутся одинаковые значения.

```
i = 10.
i = 10.
i = 10.
```

```
i = 10.
i = 10.
i = 10.
i = 10.
i = 10.
i = 10.
i = 10.
```

Можно заметить, что во втором случае вместо локальной функции используется делегат типа Action, однако это не имеет принципиального значения.

Основное отличие последнего примера от предыдущего заключается в том, что здесь сначала, в первом цикле, создаются и сохраняются в массиве действий замыкания для ряда лямбда-выражений (строка 8), но запускаются они позднее, во втором цикле (строка 12). При этом, к моменту исполнения кода из лямбда-выражений, переменная **`i`**, ссылка на которую хранится в замыканиях, уже окажется равной **`count`** (10), ведь первый цикл завершен.

Эту особенность следует учитывать и, при необходимости, передавать в замыкания изменяющиеся значения через параметр.

### События

Событие – это механизм сигнализации о выполнении некоторого действия. Для понимания сути события рассмотрим следующий пример.

Пусть имеется некоторая задача, требующая длительного решения. Например, требуется найти произведение матриц больших размеров. Для этого был написан следующий код.

```
1   using System.Diagnostics;
2
3   // Создаем генератор случайных чисел
4   var rnd = new Random();
5
6   // Задаем размеры двух исходных матриц
7   var aRows = 1000;
8   var aCols = 2000;
9   var bRows = 2000;
10  var bCols = 1000;
11
12  // Создаем две матрицы
13  var a = new double[aRows, aCols];
14  var b = new double[bRows, bCols];
15
16  // Заполняем их случайными значениями.
17  for (int i = 0; i < aRows; i++)
18  {
```

```
19      for (int j = 0; j < aCols; j++)
20      {
21          a[i, j] = rnd.NextDouble();
22          b[j, i] = rnd.NextDouble();
23      }
24  }
25
26  // Начинаем замерять время вычислений
27  Stopwatch sw = Stopwatch.StartNew();
28  // Начинаем вычисления
29  Calculator.Calc(a, b);
30  // Останавливаем замер времени
31  sw.Stop();
32
33  // Выводим на экран время вычислений
34  Console.WriteLine("Время вычислений: {0} мс.",
35                        sw.ElapsedMilliseconds);
36
37  /// <summary>
38  /// Класс, содержащий метод нахождения произведения двух матриц
39  /// </summary>
40  public static class Calculator
41  {
42
43      public static double[,] Calc(double[,] a, double[,]b)
44      {
45          // Определяем размеры исходных матриц
46          int aRows = a.GetLength(0);
47          int aCols = a.GetLength(1);
48          int bRows = b.GetLength(0);
49          int bCols = b.GetLength(1);
50          // Формируем массив, который будет содержать элементы результата
51          double[,] result = new double[aRows, bCols];
52          // Начинаем вычисления
53          for (int i = 0; i < aRows; i++)
54          {
55              for (int j = 0; j < bCols; j++)
56              {
57                  result[i, j] = 0.0;
58                  for (int k = 0; k < aCols; k++)
59                  {
60                      result[i, j] += a[i, k] * b[k, j];
61                  }
62              }
63
64          }
65          return result;
66      }
67  }
```

После того, как этот код отработает, на экране будет выведено время вычислений.

```
Время вычислений: 32184 мс.
```

Конкретные значения времени будут зависеть от мощности системы. Однако можно заметить, что в течение примерно половины минуты пользователю придется ожидать результата, глядя на пустой экран.

Чтобы показать, что процесс вычислений продвигается и дать представление о примерном времени его выполнения, можно периодически выводить на экран процент выполненных работ. Это можно сделать в одном из циклов, например, в 63 строке.

Однако здесь следует иметь в виду, что класс `Calculator` не имеет представления о том, как мы осуществляем взаимодействие с пользователем. Мы можем выводить информацию как на консоль, так и в окно Windows, либо передавать эту информацию в какие-то другие классы.

Для решения этой проблемы можно делегировать конкретный способ уведомления пользователя из метода `Calc` класса `Calculator`, какому-либо другому, внешнему методу. При этом сам класс калькулятор будет в определенное время генерировать событие и передавать в него информацию о том, сколько процентов вычислений выполнено к текущему моменту.

С этой целью перед объявлением класса потребуется добавить делегат.

```
36  // Делегат для определения события в классе
37  public delegate void NotifyHandler(float percentage);
```

А в самом классе сформировать объект-событие с помощью ключевого слова `event`.

```
44  public static event NotifyHandler OnNotify;
```

На это событие можно будет оформить подписку вне класса. Такая операция позволит задать конкретный метод или лямбда-выражение, которое будет выполняться при вызове события и называться обработчиком этого события.

```
28  // Подписываемся на событие-уведомление о ходе вычислительного процесса
29  Calculator.OnNotify +=
30      (percentage) => Console.WriteLine("Выполнено: {0}%", percentage);
```

В самом классе остается только определить место вызова события.

```
68  // На каждой пятой строчке
69  if (i % 5 == 0)
70      // уведомляем пользователя о ходе вычислительного процесса
71      OnNotify?.Invoke((float)Math.Round(100.0 * i / aRows, 1));
```

Таким образом, полный код примет следующий вид.

```
1   using System.Diagnostics;
2
3   // Создаем генератор случайных чисел
4   var rnd = new Random();
5
6   // Задаем размеры двух исходных матриц
7   var aRows = 1000;
8   var aCols = 2000;
9   var bRows = 2000;
10  var bCols = 1000;
11
12  // Создаем две матрицы
13  var a = new double[aRows, aCols];
14  var b = new double[bRows, bCols];
15
16  // Заполняем их случайными значениями.
17  for (int i = 0; i < aRows; i++)
18  {
19      for (int j = 0; j < aCols; j++)
20      {
21          a[i, j] = rnd.NextDouble();
22          b[j, i] = rnd.NextDouble();
23      }
24  }
25
26  // Начинаем замерять время вычислений
27  Stopwatch sw = Stopwatch.StartNew();
28  // Подписываемся на событие-уведомление о ходе вычислительного процесса
29  Calculator.OnNotify +=
30      (percentage) => Console.WriteLine("Выполнено: {0}%", percentage);
31  // Начинаем вычисления
32  Calculator.Calc(a, b);
33  // Останавливаем замер времени
34  sw.Stop();
35
36  // Выводим на экран время вычислений
37  Console.WriteLine("Время вычислений: {0} мс.", sw.ElapsedMilliseconds);
38
39
40  // Делегат для определения события в классе
41  public delegate void NotifyHandler(float percentage);
42  /// <summary>
43  /// Класс, содержащий метод нахождения произведения матриц
44  /// </summary>
45  public static class Calculator
46  {
47      public static event NotifyHandler OnNotify;
48      public static double[,] Calc(double[,] a, double[,]b)
49      {
50          // Определяем размеры исходных матриц
```

```
51          int aRows = a.GetLength(0);
52          int aCols = a.GetLength(1);
53          int bRows = b.GetLength(0);
54          int bCols = b.GetLength(1);
55          // Формируем массив, который будет содержать элементы результата
56          double[,] result = new double[aRows, bCols];
57          // Начинаем вычисления
58          for (int i = 0; i < aRows; i++)
59          {
60              for (int j = 0; j < bCols; j++)
61              {
62                  result[i, j] = 0.0;
63                  for (int k = 0; k < aCols; k++)
64                  {
65                      result[i, j] += a[i, k] * b[k, j];
66                  }
67              }
68              // На каждой пятой строчке
69              if (i % 5 == 0)
70                  // уведомляем пользователя о ходе вычислительного процесса
71                  OnNotify?.Invoke((float)Math.Round(100.0 * i / aRows, 1));
72          }
73          return result;
74      }
75  }
```

**Задание.** Попробуйте самостоятельно задать обработчик события в формате метода, а не лямбда-выражения.

## ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

1. Создать класс для работы с римскими числами. Реализовать в классе:
   а) поле для хранения целочисленного значения римского числа;
   б) конструкторы, позволяющие создавать римское число по строке и по значению типа int;
   в) конструктор копирования;
   г) метод перевода целочисленное значения в строку, представляющую римскую запись числа (`ToString()`);
   д) метод перевода числа в римской форме записи в числовое значение;
   е) операторы для выполнения 4-х целочисленных арифметических операций с двумя римскими числами;
   ж) операторы для выполнения 4-х целочисленных арифметических операций с римским числом и целочисленным значением (в обе стороны – 8 шт.).
   з) различные операторы сравнения двух римских чисел (6 шт.);
   и) методы проверки идентичности двух римских чисел и получение хэш-кода римского числа.
2. Создать класс для работы с обыкновенными дробями. Реализовать в классе:
   а) поля для хранения целого числителя и натурального знаменателя;
   б) свойства для получения целой части, числителя и знаменателя несократимой правильной дроби;
   в) конструктор для создания дроби по числителю и знаменателю;
   г) конструктор для создания дроби по целой части, числителю и знаменателю со значением по умолчанию для знаменателя равным 1;
   д) конструктор копирования;
   е) обеспечить контроль того, что дробь несократимая;
   ж) операторы для выполнения с дробями пяти арифметических операций (при этом вести контроль того, что числитель – целый, а знаменатель всегда натуральный);
   з) операторы унарного минуса и плюса (для симметрии);
   и) операторы сравнения двух дробей на равенство и неравенство;
   к) операторы сравнения двух дробей (больше, меньше, больше или равно, меньше или равно);
   л) оператор явного преобразования дроби к целому числу;
   м) оператор неявного преобразования дроби к вещественному числу;
   н) оператор неявного преобразования целого числа к дроби;
   о) переопределенный метод представления дроби в виде строки, формирующего правильную несократимую дробь;

п) переопределенные методы проверки эквивалентности двух дробей и получения хэш-кода.

3. Создать класс **`TrigComplex`** – наследник класса **`Complex`**, приведенном в примере выше, для работы с комплексными числами в тригонометрической форме записи.
4. Создать класс для работы с матрицами. Реализовать в классе:
   а) конструктор с параметрами для создания матрицы по двумерному массиву;
   б) конструктор для создания нулевой матрицы заданного размера;
   в) конструктор копирования;
   г) свойства для получения размеров матрицы;
   д) индексатор для получения и задания значения элемента по его индексам с проверкой возможности выполнения операции и возвращения **`null`**, при некорректных индексах;
   е) метод **`ToString()`** для получения строкового представления матрицы
   ж) метод проверки матриц на идентичность;
   з) метод получения хэш-кода для матрицы;
   и) операторы для выполнения матричных операций: сложения двух матриц, умножения матрицы на число, умножения двух матриц, деления матрицы на число; вычитания матрицы из матрицы, проверка равенства и неравенства двух матриц;
   к) свойство только для чтения для получения транспонированной матрицы;
   л) создайте события для уведомления пользователя о ходе выполнения операций над матрицами;
   м) в методе **`Main()`** приложения провести тестирование всех элементов класса.
5. Добавить в предыдущее задание новый класс-наследник «квадратная матрица». Реализовать в классе:
   а) конструктор с параметрами для создания матрицы по двумерному массиву с проверкой на то, является ли исходный массив квадратным (при необходимости выбрасывать исключительную ситуацию);
   б) конструктор копирования;
   в) конструктор нисходящего преобразования типа;
   г) конструктор для создания нулевой матрицы заданного размера;
   д) свойство для получения определителя матрицы;
   е) свойство для получения обратной матрицы (при невозможности ее вычислить, выбрасывать исключительную ситуацию) с поддержкой события о ходе выполнения вычислительного процесса;

ж) создать классы исключительных ситуаций для работы с собственными исключениями;

з) в методе `Main()` приложения провести тестирование всех элементов класса.

6. Создайте обобщенные классы `Pair` и `Triplet`, предназначенные для хранения пар и троек значений.
7. Создайте класс-перечислимый тип для работы с простыми числами.
8. Создайте систему типов для формирования на заданном отрезке набора узлов.

А. Создать абстрактный класс `Nodes` для распределения некоторым образом (абстрактно) $n$ точек (узлов) на отрезке$[a, b]$. В классе реализовать интерфейс `IReadOnlyCollection` или `IReadOnlyCollection<T>`.

Б. Добавить класс-наследник класса `Nodes` – `EqvNodes` для распределения $n$ равноотстоящих узлов на заданном отрезке. Реализовать итератор с помощью класса, реализующего интерфейс `IEnumerator` или `IEnumerator<T>`.

В. Добавить класс `ChebNodes` – наследником `Nodes` для работы с узлами Чебышева. Реализовать итератор внутри метода `GetEnumerator()` с использованием оператора `yield return`.

Для вычисления узлов Чебышева на отрезке $[-1; 1]$ можно воспользоваться формулой:

$$x_k = \cos\frac{(2k+1)\pi}{2(n+1)}, \qquad k = 0, 1, \dots, n.$$

Для вычисления узлов Чебышева на произвольном отрезке $[a; b]$ аналогичная формула примет вид:

$$x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\frac{(2k+1)\pi}{2(n+1)}, \qquad k = 0, 1, \dots, n.$$

Г. Создать класс-наследник `Nodes` – `RandNodes` для разбиения отрезка на части случайным образом, так, чтобы узлы при этом не повторялись, а при неоднократном доступе к узлу по его номеру, возвращалось бы одно и то же случайное число.

Д. Добавить класс `FixedNodes` (наследник `Nodes`) для задания набора узлов на отрезке вручную, пользователем. Класс должен реализовать интерфейс `ICollection` или `ICollection<T>`. Реализовать в классе методы добавления и удаления узлов.

# II. ОСНОВЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В C#

## 1. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ПАРАЛЛЕЛЬНЫХ ПОТОКОВ

### Класс Thread

Постоянное совершенствование многопроцессорных систем и увеличение числа ядер в современных процессорах сделали параллельные алгоритмы весьма актуальными. Использование нескольких вычислительных устройств одновременно позволяет во многих случаях повысить скорость выполнения трудоемких операций по сравнению с выполнением той же задачи на одном процессоре.

---

Следует иметь в виду, что некоторые действия легко поддаются распараллеливанию, а некоторые — нет. Так, легко распараллелить такие задачи, как, например, поиск суммы и произведения большого количества чисел, умножения двух матриц, определения простых чисел из определенного диапазона. В таких задачах итерации являются независимыми от других. Однако решение многих задач не поддается распараллеливанию или требуют применения определенных методик. Как правило, в таких случаях на каждой итерации циклов требуются результаты, полученные на предыдущих этапах. В качестве примеров здесь можно привести многие алгоритмы поиска значения числа π, численные методы решения нелинейных уравнений, некоторые рекурсивные алгоритмы и т.п.

---

Для распределения нагрузки между разными вычислительными устройствами (ядрами процессора) необходимо распределить вычисления между разными потоками. Поток – это независимый путь исполнения программного кода, способный выполняться одновременно с другими потоками. Обычно при запуске приложения, операционной системой формируется процесс, в котором автоматически создается так называемый главный поток. Все действия выполняются одним потоком строго последовательно. Для выполнения нескольких действий параллельно (например, на разных ядрах процессора, или, хотя бы, на одном процессоре в режиме разделения времени), необходимо добавить в процесс другие потоки.

Базовым классом для создания в программе дополнительных потоков является класс `Thread`. Он создает и контролирует поток, задает приоритет и возвращает его статус.

Создать поток можно при помощи одного из конструкторов класса. Подробная информация о них приведена в таблице II-1.

**Таблица II-1**

**Конструкторы класса Thread**

| Конструкторы | Описания |
|---|---|
| `Thread(ThreadStart start)` | Инициализирует новый экземпляр класса `Thread`. Делегат `ThreadStart start` указывает на метод, который вызывается при запуске потока. |
| `Thread( ParameterizedThread-Start start)` | Инициализирует новый экземпляр класса `Thread`. При этом указывается делегат `Parameterized-ThreadStart start`, позволяющий одному произвольному объекту быть переданным в поток при его запуске. |

---

**Замечание.** Вместо объектов `ThreadStart` и `ParameterizedThreadStart` допускается передача в конструктор лямбда-выражений, тело которых будет исполнено в отдельном потоке.

---

Следующая таблица содержит список основных свойств класса `Thread`.

**Таблица II-2**

**Свойства класса Thread**

| Свойства | Описания |
|---|---|
| `bool IsAlive` | Возвращает значение, показывающее статус выполнения текущего потока. Свойство содержит значение `true`, если этот поток был запущен и не был завершен; в противном случае свойство содержит значение `false`. |
| `bool IsBackground` | Получает или задает значение, показывающее, является ли поток фоновым. Фоновые потоки будут завершаться принудительно при завершении всех остальных (не фоновых) потоков приложения. |
| `static Thread CurrentThread` | Статическое свойство, позволяющее получить ссылку на текущий поток. |
| `int ManagedThreadId` | Позволяет получить числовой идентификатор текущего потока. |

| Свойства | Описания |
|---|---|
| `string Name` | Содержит имя потока. |
| `ThreadPriority Priority` | Получает или задает значение, указывающее на планируемый приоритет потока.<br>Значение свойства может быть одним из следующих:<br>*Lowest:*<br>Низший приоритет. Выполнение потока производится в последнюю очередь.<br>*BelowNormal:*<br>Приоритет ниже нормального. Выполнение производится после потоков с приоритетом Normal и до потоков с приоритетом Lowest.<br>*Normal:*<br>Нормальный приоритет. Выполнение производится после потоков с приоритетом AboveNormal и до потоков с приоритетом BelowNormal. Это значение установлено по умолчанию.<br>*AboveNormal:*<br>Приоритет потока выше нормального. Выполнение производится после потоков с приоритетом Highest и до потоков с приоритетом Normal.<br>*Highest:*<br>Наивысший приоритет. Выполнение потока производится в первую очередь. |
| `ThreadState` | Возвращает значение, содержащее состояние текущего потока. Оно может принимать одно из значений:<br>*Running:*<br>Поток был запущен, не заблокирован, и нет ожидающего исключения `ThreadAbortException`.<br>*StopRequested:* |

| Свойства | Описания |
|---|---|
| | Поток получает запрос на остановку. Предназначено только для внутреннего использования.<br>*SuspendRequested:*<br>Запрашивается приостановка работы потока.<br>*Background:*<br>Поток выполняется как фоновый, в противоположность потокам переднего плана. Это состояние управляется заданием свойства `Thread.IsBackground`.<br>*Unstarted:*<br>Метод `Start()` не был вызван для потока.<br>*Stopped:*<br>Поток был остановлен.<br>*WaitSleepJoin:*<br>Поток заблокирован. Это может произойти в результате вызова методов `Thread.Sleep` или `Thread.Join`, а также в результате запроса блокировки, например при вызове методов `Monitor.Enter` или `Monitor.Wait`, или в результате ожидания объекта синхронизации потока, такого как `ManualResetEvent`.<br>*Suspended:*<br>Поток был приостановлен. |

**Таблица II-3**

**Методы класса Thread**

| Методы | Описания |
|---|---|
| `void Join()` | Блокирует вызывающий поток до завершения этого потока, продолжая отправлять стандартные сообщения. |

| Методы | Описания |
|---|---|
| `static void Sleep(int millisecondsTimeout)` | Блокирует текущий поток на заданное количество миллисекунд. |
| `static void Sleep(int MillisecondsTimeout)` | Статический метод, позволяющий приостановить выполнение потока на заданное количество миллисекунд. |
| `void Start()` | Запускает поток на выполнение. |
| `void Start(Object parameter)` | Запускает поток на выполнение, а также передает объект с данными методу, исполняемому в потоке. |

**Обратите внимание,** что в классе `Thread` присутствуют также некоторые методы, которые позволяют прекращать, приостанавливать или возобновлять выполнение потока. Тем не менее, такие методы не рекомендуются к использованию, поскольку их применение может негативно сказаться на предсказуемости работы приложения. При этом для выполнения подобного рода операций необходимо их реализовывать средствами самого потока, используя, например, различные переменные-флаги.

Рассмотрим код программы, демонстрирующий некоторые методы работы с потоками. В данном примере показывается чередование работы двух параллельных потоков при выводе данных на экран.

```
1   // Формируем поток, который будет выводить символ "a"
2   Thread t1 = new Thread(WriteA);
3   // Формируем второй поток, который будет выводить заданную в параметре строку
4   var ts = new ParameterizedThreadStart(WriteStr);
5   // Формируем объект "Поток", которому передаем параметры запуска
6   Thread t2 = new Thread(ts);
7   // Запускаем поток 1, который будет выводить символ "a"
8   t1.Start();
9   // Запускаем поток 2, который будет выводить символ "b"
10  t2.Start("b");
11
12
13  // Метод выводит 100 букв "a"
14  void WriteA()
15  {
16      for (int i = 0; i < 100; ++i)
17      {
18          Console.Write("a");
19      }
```

```
}

// Метод выводит 100 раз строку str, переданную через параметр
void WriteStr(object? str)
{
    for (int i = 0; i < 100; ++i)
    {
        Console.Write(str);
    }
}
```

Примерный результат выполнения данного кода показан ниже:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb
bbbbbbbb
```

Видно чередование вывода разных символов, что свидетельствует о переменной работе двух разных потоков.

Еще один фрагмент программы показывает вариант создания потоков с применением лямбда-выражений и использование некоторых дополнительных методов и свойств класса `Thread`. Также этот пример наглядно иллюстрирует параллельную работу запускаемых на выполнение потоков.

```
using System.Diagnostics;

Console.WriteLine(
    "Начало работы. Поток {0} (основной)",
    Thread.CurrentThread.ManagedThreadId
);
Random rnd = new Random();

const int threadCount = 10;
Thread[] threads = new Thread[threadCount];

// Начинаем замер времени выполнения программы
Stopwatch sw = Stopwatch.StartNew();

// Формируем потоки для их последующего запуска
for (int i = 0; i < threadCount; i++)
{
    threads[i] = new Thread(() => {
        Console.WriteLine(
            "Начало работы потока №{0}",
            Thread.CurrentThread.ManagedThreadId
        );
        // Генерируем случайное число от 10 до 100
        var t = rnd.Next(10, 100);
        // Выдерживаем паузу случайной величины
        Thread.Sleep(t);
```

```
27          // Выводим информацию о полученном значении
28          Console.WriteLine(
29              "Поток {0} ожидал {1}мс.",
30              Thread.CurrentThread.ManagedThreadId,
31              t
32          );
33      });
34  }
35
36  // Производим параллельный запуск созданных потоков
37  for (int i = 0;i < threadCount; i++)
38  {
39      threads[i].Start();
40  }
41
42  // Дожидаемся завершения всех созданных дополнительных потоков
43  for (int i = 0;i <threadCount; i++)
44  {
45      threads[i].Join();
46  }
47
48  // Останавливаем замер времени
49  sw.Stop();
50
51  Console.WriteLine(
52      "Время выполнения приложения: {0}мс.",
53      sw.ElapsedMilliseconds
54  );
55  Console.WriteLine(
56      "Завершение потока {0}",
57      Thread.CurrentThread.ManagedThreadId
58  );
```

В приведенном примере создается 10 дополнительных потоков, каждый из которых генерирует некоторое случайное число от 10 до 100, и затем приостанавливает свое выполнение на полученное число миллисекунд.

В результате выполнения программы на экран будет выведен примерно следующий текст.

```
Начало работы. Поток 1 (основной)
Начало работы потока №8
Начало работы потока №9
Начало работы потока №10
Начало работы потока №11
Начало работы потока №12
Начало работы потока №13
Начало работы потока №14
```

```
Начало работы потока №15
Начало работы потока №16
Начало работы потока №17
Поток 15 ожидал 22мс.
Поток 16 ожидал 26мс.
Поток 11 ожидал 20мс.
Поток 13 ожидал 46мс.
Поток 12 ожидал 37мс.
Поток 8 ожидал 55мс.
Поток 14 ожидал 54мс.
Поток 9 ожидал 57мс.
Поток 17 ожидал 76мс.
Поток 10 ожидал 93мс.
Время выполнения приложения: 99мс.
Завершение потока 1
```

Видно, что суммарное время пауз в отдельных потоках намного больше общего времени выполнения приложения, что доказывает параллельность выполнения созданных дополнительных потоков.

---

**Обратите внимание!** Создание потока – дорогостоящая операция, которая расходует много системных ресурсов. Поэтому в программе не следует создавать слишком большое количество потоков одновременно. Обычно достаточно ограничиться количеством, сопоставимым с числом процессоров (ядер процессора), имеющихся в вычислительной системе. В случае необходимости использования большего числа потоков, стоит подумать об использовании иных инструментов, например, задач, которые будут рассмотрены позже.

---

### Синхронизация потоков

При разработке параллельных алгоритмов требуется каким-то образом организовать связь между различными потоками, что, например, необходимо для получения общего результата. Для этого можно воспользоваться одним из двух вариантов обмена данными: общей памятью или системой передачи сообщений.

В системах передачи сообщений применяются так называемые каналы и блоки сообщений. Системы с общей памятью требуют введения блокировок для обрабатываемых данных.

---

Следует учесть, что в некоторых ситуациях (при неверной организации вычислений) параллельный код может выполняться даже медленнее, чем последовательный, из-за дополнительной нагрузки, оказываемой на систему. Например, такое может произойти, когда создается большое число потоков, но при этом каждый из них производит небольшой объем работы.

---

В примерах, приведенных ниже, будет разобран случай использования общей памяти.

Потоки, работающие параллельно, могут пытаться обращаться совместно к одним и тем же ресурсам, например, два потока могут одновременно производить изменения одной и той же переменной, что может привести к неправильной работе приложения. Чтобы избежать ошибок, необходимо контролировать доступ к общим для нескольких потоков ресурсам. Обычно в таких случаях используются те или иные механизмы синхронизации, которые реализованы в специальных классах, входящих в библиотеку .NET. Перечень таких классов приведен в таблице II-4.

**Таблица II-4**

**Классы, используемые для синхронизации потоков**

| Класс | Описание |
|---|---|
| AutoResetEvent | Блокирует поток до установки события другим потоком. |
| ManualResetEvent | Блокирует один или несколько потоков до установления события другим потоком. |
| Interlocked | Организует совместную работу потоков при выполнении простых операций с числами: инкремент, декремент, обмен и сравнение. |
| Monitor | Гарантирует последовательный доступ нескольких потоков к заблокированному объекту. |
| Mutex | Гарантирует последовательный доступ нескольких потоков к заблокированному объекту и, в отличие от Monitor, может быть использован для синхронизации работы разных приложений. |
| ReaderWriterLock | Обеспечивает параллельное чтение ресурса несколькими потоками, но строго последовательную запись информации в ресурс потоками поочередно. |

В настоящем пособии будут рассмотрены более подробно возможности одного из этих классов – «**`Monitor`**». Он позволяет осуществлять синхронизацию потоков на уровне доступа к объектам. Все методы класса (см. таблицу II-5) являются статическими и не требуют создания его экземпляров.

**Таблица II-5**

**Класс Monitor**

| Методы | Описания |
|---|---|
| `static void Enter( Object obj)` | Получает эксклюзивную блокировку указанного объекта. Параметр **obj** – объект ссылочного типа, для которого получается блокировка монитора. |
| `static bool TryEnter (Object obj, int millisecondsTimeout)` | Метод пытается получить эксклюзивную блокировку указанного объекта. Параметр **obj** – объект ссылочного типа, для которого получается блокировка монитора. Если по истечении указанного в миллисекундах времени блокировка не была получена, метод возвращает **false** и передает управление в поток, позволяя ему выполнить другие операции. В противном случае метод возвращает значение **true**. |
| `static void Exit(Object obj)` | Освобождает эксклюзивную блокировку указанного объекта. Параметр **obj** – объект ссылочного типа, для которого снимается блокировка монитора, должен быть ранее заблокирован методом **Enter**. |
| `static void Pulse( Object obj)` | Уведомляет первый поток в очереди об изменении состояния объекта с блокировкой. Параметр **obj** – это объект ссылочного типа, которого ожидает поток. |
| `static void PulseAll (Object obj)` | Уведомляет все ожидающие потоки об изменении состояния объекта. Параметр **obj** – это объект ссылочного типа, которого ожидают потоки. |
| `static bool Wait (Object obj)` | Освобождает блокировку объекта **obj** и блокирует текущий поток до тех пор, пока тот не получит блокировку снова. Возвращаемое значение – **true**, если вызывающий поток заново получил блокировку заданного объекта. Этот метод не осуществляет возврат, если блокировка вновь не получена. |
| `static bool Wait( Object obj,` | Освобождает блокировку объекта **obj** и блокирует текущий поток до тех пор, пока тот не получит блокировку снова. Если указанный |

| Методы | Описания |
|---|---|
| `int millisec-ondsTimeout)` | временной интервал (`millisecondsTimeout`) истекает, поток встает в очередь готовности. |

Рассмотрим два примера, демонстрирующих необходимость выполнения синхронизации потоков.

***Пример 1.*** Пусть имеется класс, представляющий собой простейшую реализацию банковского счета. На этот счет можно положить или снять с него определенную сумму средств. При этом, желаемую сумму можно снять со счета только при наличии необходимого объема денег у клиента. Допустим, что примерно в одно время работодатель зачисляет на счет клиенту заработную плату, а клиент пытается снять сумму, превышающую первоначальный баланс со счета. Поскольку в реальности эти операции очевидно будут выполняться на разных устройствах, смоделируем это с помощью различных, параллельно работающих потоков программы. В итоге код приложения может принять следующий вид.

```
1   var ba = new BankAccount(50000f);
2   new Thread(() =>
3   {
4           // Поток, моделирующий работу со стороны работодателя (пополнение счета)
5           Console.WriteLine("Поток {0}. Пополняем счёт.",
6               Thread.CurrentThread.ManagedThreadId);
7           ba.MakeTransaction(50000f);
8           Console.WriteLine("Поток {0}. {1}",
9               Thread.CurrentThread.ManagedThreadId, ba);
10  }).Start();
11  try
12  {
13          // Главный поток моделирует работу со стороны банкомата (снятие со счета)
14          Console.WriteLine("Поток {0}. {1}",
15          Thread.CurrentThread.ManagedThreadId, ba);
16          Console.WriteLine("Поток {0}. Снимаем со счёта.",
17              Thread.CurrentThread.ManagedThreadId);
18          ba.MakeTransaction(-75000f);
19  } catch (Exception ex)
20  {
21          // В случае недостаточности средств, сообщаем об ошибке
22          Console.WriteLine(ex.Message);
23          Console.WriteLine("Err: Поток {0}. {1}",
24              Thread.CurrentThread.ManagedThreadId, ba);
25  }
26
27  // Ожидаем завершения работы всех потоков
28  Thread.Sleep(1000);
29
```

```
30  /// <summary>
31  /// Класс банковского счета.
32  /// </summary>
33  public class BankAccount
34  {
35      // Сумма, доступная клиенту
36      public float Sum { get; set; }
37
38      // Создание счета с заданием исходной хранимой суммы
39      public BankAccount(float initSum)
40      {
41          Sum = initSum;
42      }
43
44      // Выполнение банковской транзакции:
45      public void MakeTransaction(float sum_delta)
46      {
47          float s = Sum;
48          // Проверяем, что на счете достаточно средств для снятия:
49          if (sum_delta < 0 && s + sum_delta >= 0f ||
50              sum_delta >= 0f)
51          {
52              Sum = s + sum_delta;
53          }
54          else
55          {
56              // Если денег недостаточно, выбрасываем исключение:
57              throw new Exception("Недостаточно средств на счете " +
58                  $"для снятия {-sum_delta}р.");
59          }
60      }
61
62      // Получение информации о счете в виде строки:
63      public override string ToString() => $"На счёте {Sum}р.";
64  }
```

Данный код может генерировать различные случаи, что зависит от того, в каком порядке сработают параллельно работающие потоки. Одним из вероятных вариантов может оказаться следующий.

```
Поток 1. На счёте 50000р.
Поток 1. Снимаем со счёта.
Поток 7. Пополняем счёт.
Поток 7. На счёте 100000р.
Недостаточно средств на счете для снятия 75000р.
Err: Поток 1. На счёте 100000р.
```

Можно заметить, что приложение ведет себя не вполне адекватно: несмотря на то, что счет пополняется на необходимую сумму, выдается сообщение о недостаточности средств. Это происходит из-за неконтролируемой

последовательности действий при совместном доступе потоков к общему ресурсу – переменной, хранящей сумму на счете.

В приведенном случае, очевидно, произошло следующее.

- Поток №1 выводит информацию о балансе счета (50000р.) и начинает операцию снятия. Он узнает, остаток на счете и запоминает его в локальной переменной.
- В это время в работу включается Поток №7, который выполняет операцию пополнения счета на 50000р. Теперь счет содержит необходимую сумму (100000р.) и клиенту, очевидно, доступно снятие 75000р.
- Поток №1 проверяет условия возможности снятия, используя для этого ранее запомненное старое значение баланса счета, и, поскольку тот баланс недостаточен, выбрасывает исключение. Выводится сообщение об ошибке и, затем, актуальное состояние счета – 100000р. Клиент недоумевает от полученных сообщений!

Чтобы исправить ситуацию, потребуется выполнить синхронизацию работы потоков. Необходимо сделать так, чтобы после начала транзакции одним потоком, другой не имел возможности работать со счетом, пока первый не закончит свою работу. Это можно реализовать при помощи класса `Monitor` и его статических методов `Enter` и `Exit`, открывающих и завершающих синхронизированную секцию. В качестве параметра этим методам передается объект ссылочного типа, который является общим для нескольких потоков. Такую конструкцию также можно заменить использованием ключевого слова `lock` с таким же параметром. При этом, чтобы синхронизация работала, блок нужно создавать во всех местах, где производится работа с общим объектом.

```
1   var ba = new BankAccount(50000f);
2   new Thread(() =>
3   {
4       Monitor.Enter(ba);
5       try
6       {
7           // Поток, моделирующий работу со стороны работодателя (пополнение счета)
8           Console.WriteLine("Поток {0}. Пополняем счёт.",
9               Thread.CurrentThread.ManagedThreadId);
10          ba.MakeTransaction(50000f);
11          Console.WriteLine("Поток {0}. {1}",
12              Thread.CurrentThread.ManagedThreadId, ba);
13      }
14      finally { Monitor.Exit(ba); }
15  }).Start();
16  try
```

```
{
    lock (ba)
    {
        // Главный поток моделирует работу со стороны банкомата (снятие со счета)
        Console.WriteLine("Поток {0}. {1}",
        Thread.CurrentThread.ManagedThreadId, ba);
        Console.WriteLine("Поток {0}. Снимаем со счёта.",
            Thread.CurrentThread.ManagedThreadId);
        ba.MakeTransaction(-75000f);
    }
} catch (Exception ex)
{
    lock (ba)
    {
        // В случае недостаточности средств, сообщаем об ошибке
        Console.WriteLine(ex.Message);
        Console.WriteLine("Err: Поток {0}. {1}",
            Thread.CurrentThread.ManagedThreadId, ba);
    }
}

// Ожидаем завершения работы всех потоков
Thread.Sleep(1000);
```

После внесения изменений вывод на экране уже будет более логичным.

```
Поток 1. На счёте 50000р.
Поток 1. Снимаем со счёта.
Недостаточно средств на счете для снятия 75000р.
Err: Поток 1. На счёте 50000р.
Поток 7. Пополняем счёт.
Поток 7. На счёте 100000р.
```

Видно, что поток 1 полностью завершает свою работу до того, как приступает к работе поток 7 и отображаемые балансы соответствуют порядку выполнения операций.

***Пример 2.*** Во втором примере требовалось распараллелить вычисление суммы множества первых натуральных чисел. С этой целью было создано несколько потоков, каждый из которых прибавлял к общей сумме свою часть слагаемых.

```
Console.WriteLine(Summator.Sum(1000000));
static class Summator
{
    /// <summary>
    /// Параллельное вычисление суммы n первых натуральных чисел
    /// </summary>
    /// <param name="n">Количество суммируемых натуральных чисел</param>
    /// <returns>Сумма n первых натуральных чисел</returns>
    public static long Sum(int n)
```

```
    {
        // Разделим вычисления для примера на 4 потока:
        var threadCount = 4;
        // Вычислим число слагаемых в каждой частичной сумме:
        var size = n / threadCount;
        // Переменная для хранения итогового результата:
        var resultSum = 0L;
        // Список потоков:
        var threads = new List<Thread>();
        // Для каждой части слагаемых:
        for (int threadId = 1; threadId <= threadCount; threadId++)
        {
            // Формируем потоки для вычисления и добавляем их в список:
            threads.Add(new Thread((tId) =>
            {

                // Считаем частичную сумму в каждом потоке
                for (int i = (int)tId; i <= n; i += threadCount)
                    resultSum += i;

            }));
            // Производим запуск последнего сформированного потока:
            threads.Last().Start(threadId);
        }
        // Дожидаемся завершения всех потоков из списка:
        foreach (var thread in threads) thread.Join();
        // Возвращаем полученный результат:
        return resultSum;
    }
}
```

В результате выполнения программы были получены следующие результаты.

| Количество слагаемых | Полученная сумма |
|---|---|
| 10 | 55 |
| 100 | 5050 |
| 1000 | 500500 |
| 10000 | 50005000 |
| 100000 | 5000050000 |
| 1000000 | **335827188158** |
| 10000000 | **23169232974710** |

Можно видеть, что до некоторых пор результаты были корректными. Однако с ростом числа слагаемых возрастала вероятность одновременного обращения нескольких потоков к общей переменной, что в результате стало приводить к ошибкам вычислений.

Добавим в программный код синхронизацию, внеся изменения в 28 строку.

```
28  lock (threads) { resultSum += i; }
```

После этого изменения результаты станут корректными, поскольку обращаться к общей переменной в один момент времени сможет только один поток.

| Количество слагаемых | Полученная сумма |
|---|---|
| 1000000 | 500000500000 |
| 10000000 | 50000005000000 |

Тем не менее, если посмотреть, сколько времени тратится на вычисления, то можно заметить, что процесс сильно замедлился (примерно в 4 раза). Это происходит из-за того, в ходе выполнения цикла мы на каждой итерации «убиваем» параллельность, и полностью сводим «на нет» эффект от создания нескольких потоков.

---

**Обратите внимание!** Синхронизированные блоки должны содержать минимум трудоемких операций и выполняться относительно редко.

---

Для поддержания оптимального уровня быстродействия перепишем код в потоках так, как показано ниже.

```
22  // Формируем потоки для вычисления и добавляем их в список:
23  threads.Add(new Thread((tId) =>
24  {
25      var s = 0L;
26      // Считаем частичную сумму в каждом потоке
27      for (int i = (int)tId; i <= n; i += threadCount)
28          s += i;
29      lock (threads) { resultSum += s; }
30  }));
```

В таком варианте каждый поток сначала вычисляет свою часть суммы в локальной переменной `s`, и лишь в конце своей работы объединяет полученные промежуточные результаты в общую сумму. Таким образом, количество попаданий в синхронизированный блок уменьшается многократно, что, в свою очередь, приводит к восстановлению производительности всей программы.

### Приостановка работы потоков

При решении ряда задач требуется блокировать выполнение потока до наступления некоторого события.

Предположим, что имеется некоторая переменная, доступ к которой есть у двух потоков. Один из них производит запись значений в общую переменную, а другой – чтение из неё. Если первому потоку требуется достаточно длительное время для генерации нового значения, то второй поток должен ожидать, пока это значение не появится, прежде чем он сможет продолжить свою работу. Схема взаимодействия таких потоков показана на рисунке 3.

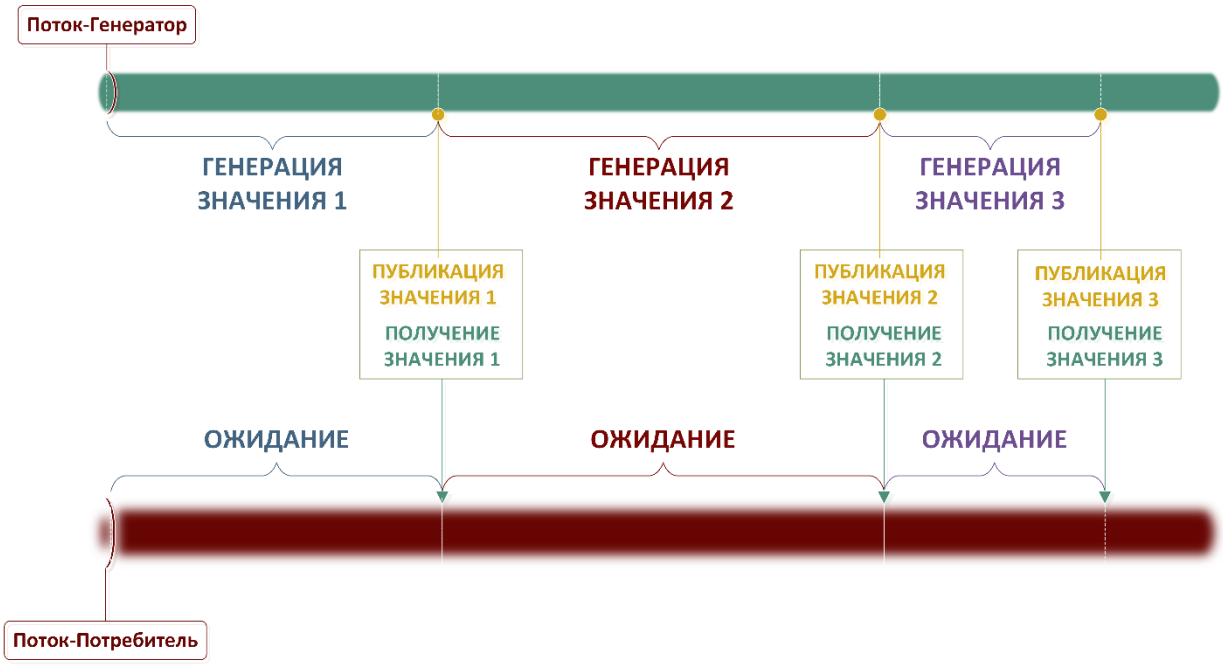


Рис. 3. Схема работы потоков по принципу «Поставщик-потребитель»

Для организации такой работы можно воспользоваться парой методов `Wait()` и `Pulse()` из класса `Monitor`. При необходимости приостановить работу потока, в нем вызывается метод `Wait()`. Тогда его выполнение будет приостановлено до тех пор, пока в каком-либо другом потоке не будет вызван метод `Pulse()`.

---

**Обратите внимание.** Вызов методов `Wait()` и `Pulse()` возможен только внутри синхронизированного блока (то есть между вызовами `Monitor.Enter()` и `Monitor.Exit()`) для одного и того же объекта синхронизации.

**Обратите внимание.** При приостановке работы потока, для него снимается эксклюзивная блокировка объекта синхронизации.

---

Рассмотрим пример, демонстрирующий взаимодействие нескольких потоков при работе с объектом класса блокирующей очереди. Обычно класс очереди выбрасывает исключение при попытке получения значения из пустой коллекции. В отличие от него блокирующая очередь будет

приостанавливать поток-получатель значений, пока поток-генератор значений не добавит в очередь новый элемент.

```
var rnd = new Random();
// Создание блокирующей очереди
var queue = new BlockingQueue<int>();
// Создаем массив потоков
var threads = new List<Thread>
{
    // Первый поток-получатель элементов
    new Thread(Receive),
    // Второй поток-получатель элементов
    new Thread(Receive),
    // Поток-генератор элементов
    new Thread(() =>
        {
            while(true)
            {
                Thread.Sleep(rnd.Next(100, 3001));
                var val = rnd.Next(1, 101);
                Console.WriteLine(
                    $"Поток {Thread.CurrentThread.ManagedThreadId} " +
                    $"создал значение {val}"
                );
                queue.Enqueue(val);
            }
        })
};

// Запуск потоков на выполнение в качестве фоновых.
// (Фоновые потоки будут завершены при завершении главного потока приложения).
threads.ForEach(t => { t.IsBackground = true; t.Start(); });

Thread.Sleep(10000);

// Метод получения элемента из очереди
void Receive()
{
    while (true)
    {
        // Получаем элемент. Если элемент отсутствует, работа метода приостанавливается.
        var val = queue.Dequeue();
        Console.WriteLine(
            $"Поток {Thread.CurrentThread.ManagedThreadId} " +
            $"получил значение {val}"
        );
    }
}

/// <summary>
```

```
48  /// Класс блокирующей очереди основан на классе "Очередь", однако
49  /// при попытке получения элемента из пустой очереди не выбрасывает исключение,
50  /// а замораживает выполнение потока до тех пор, пока в очереди не появятся новые элементы.
51  /// </summary>
52  /// <typeparam name="T">Тип элементов очереди</typeparam>
53  public class BlockingQueue<T> : Queue<T>
54  {
55      /// <summary>
56      /// Конструктор очереди по умолчанию
57      /// </summary>
58      public BlockingQueue() : base() { }
59      /// <summary>
60      /// Конструктор очереди, позволяющий задать начальный размер
61      /// </summary>
62      /// <param name="capacity">Начальный размер очереди</param>
63      public BlockingQueue(int capacity) : base(capacity) { }
64      /// <summary>
65      /// Конструктор очереди по набору перечислимых элементов
66      /// </summary>
67      /// <param name="items">Элементы очереди</param>
68      public BlockingQueue(IEnumerable<T> items) : base(items) { }
69
70      /// <summary>
71      /// Метод позволяет получить элемент из очереди.
72      /// В случае отсутствия элемента в очереди, поток, вызвавший метод, будет заморожен
73      /// до появления нового элемента
74      /// </summary>
75      /// <returns>Элемент, расположенный в начале очереди</returns>
76      public new T Dequeue()
77      {
78          lock (this)
79          {
80              // Пока в очереди нет элементов
81              while (Count == 0)
82              {
83                  // Ожидаем их появления
84                  Monitor.Wait(this);
85              }
86              // Элемент появился, можно извлекать...
87              return base.Dequeue();
88          }
89      }
90
91      /// <summary>
92      /// Метод добавления элемента в очередь.
93      /// После появления нового элемента, выполняется уведомление ожидающего потока о
94      /// возможности продолжения его работы.
95      /// </summary>
96      /// <param name="value">Добавляемый элемент</param>
97      public new void Enqueue(T value)
98      {
```

```
99          lock (this)
100         {
101             // Добавляем новый элемент в очередь
102             base.Enqueue(value);
103             // И уведомляем ожидающий поток о возможности продолжения работы
104             Monitor.Pulse(this);
105         }
106     }
107 }
```

При выполнении этой программы на экран будет выводиться примерно следующее.

```
Поток 9 создал значение 27
Поток 7 получил значение 27
Поток 9 создал значение 11
Поток 8 получил значение 11
Поток 9 создал значение 68
Поток 7 получил значение 68
Поток 9 создал значение 82
Поток 8 получил значение 82
Поток 9 создал значение 79
Поток 7 получил значение 79
Поток 9 создал значение 71
Поток 8 получил значение 71
```

## 2. БИБЛИОТЕКА TASK PARALLEL LIBRARY (TPL)

### Задачи. Класс Task

Рассмотренный в предыдущих разделах класс `Thread` предоставляет базовую функциональность для создания многопоточных приложений. Более современным подходом считается использование библиотек параллельных задач, TPL (Task Parallel Library). Данная библиотека упрощает работу с многоядерными системами и оптимизирует работу с потоками; в целом она предназначена для повышения эффективности написания программ за счет упрощения процесса добавления параллелизма в приложения. Так же TPL позволяет динамически масштабировать степень параллелизма для наиболее эффективного использования всех доступных процессоров.

Базовым классом в этой библиотеке является класс `Task` (задача). Он представляет собой некоторую продолжительную задачу, которая выполняется асинхронно в пуле потоков. Это позволяет сократить нежелательные расходы ресурсов на создание новых потоков и переиспользовать потоки, которые закончили выполнять назначенную им работу.

Есть несколько альтернативных способов создать задачу, которые приведены в следующем фрагменте кода.

```
// Создание задачи с действием (Action) без параметров
Task t1 = Task.Run(() => { /* Действия задачи... */ });
// Создание задачи с действием, принимающим параметр, с последующим её запуском
Task t2 = new Task((state) => { /* Действия задачи... */ }, 2);
t2.Start();
// Создание задачи с функцией, принимающей параметр и возвращающей результат
Task<string> t3 = Task<string>.Factory.StartNew((n) => {
    /* Действия задачи... */
    return $"Задача №{n}";
}, 3);
Console.WriteLine(t3.Result);
```

При использовании методов `Run()` и фабричного метода `StartNew()` задача запускается на выполнение сразу после создания. При использовании же конструктора класса `Task`, запуск можно произвести позднее, вызвав для этого метод `Start()`.

Для того, чтобы дождаться завершения задачи можно использовать метод `Wait()`. В случае, когда вызов задачи предполагает возврат значения, для тех же целей можно обратиться к ее свойству `Result`. Такое обращение позволит получить результат выполнения задачи, который будет готов после ее завершения.

Множество задач можно разместить в массиве. При этом появится возможность дождаться завершения всех этих задач при помощи статического метода `WaitAll()`, либо завершения хотя бы одной из них при помощи метода `WaitAny()`.

```
var tasks1 = new[] {
    new Task(() => { Thread.Sleep(300); Console.WriteLine("1"); }),
    new Task(() => { Thread.Sleep(200); Console.WriteLine("2"); }),
    new Task(() => { Thread.Sleep(100); Console.WriteLine("3"); }),
};
foreach (var task in tasks1) task.Start();
Task.WaitAll(tasks1);
Console.WriteLine("Все задачи (1, 2, 3) завершены.");

var tasks2 = new[] {
    new Task(() => { Thread.Sleep(300); Console.WriteLine("4"); }),
    new Task(() => { Thread.Sleep(200); Console.WriteLine("5"); }),
    new Task(() => { Thread.Sleep(100); Console.WriteLine("6"); }),
};
foreach (var task in tasks2) task.Start();
Task.WaitAny(tasks2);
Console.WriteLine("Одна из задач (4, 5, 6) завершена.");
```

Выполнение этого фрагмента кода приведет к появлению на экране следующих сообщений.

```
3
2
1
Все задачи (1, 2, 3) завершены.
6
Одна из задач (4, 5, 6) завершена.
```

Задачи также могут быть вложенными друг в друга. Также у программиста есть возможность запускать одни задачи по окончанию выполнения других. Такие задачи называются задачами продолжения. В следующем примере показано, как запустить несколько задач внутри другой, получить их результаты и воспользоваться ими в задаче продолжения.

```
1   var parentTask = Task<int[]>.Factory.StartNew(() => {
2       foreach (var task in tasks3) task.Start();
3       var results = new List<int>();
4       foreach (var t in tasks3) results.Add(t.Result);
5       return results.ToArray();
6   });
7   var continuationTask = parentTask.ContinueWith(task => {
8       Console.WriteLine("Сумма результатов: {0}", task.Result.Sum());
9   });
10  continuationTask.Wait();
```

Результатом выполнения такого кода будет следующая строка на экране.

```
Сумма результатов: 24
```

Рассмотрим более комплексный пример, который показывает некоторые возможности применения задач и их преимущества по сравнению со стандартными потоками.

Пусть имеется две матрицы, которые нужно перемножить. Для лучшего понимания приведем сначала программный код, содержащий класс, в котором эта операция реализована вообще без применения многопоточности.

```
1   using Parallel;
2   using System.Diagnostics;
3
4   // Размеры матрицы
5   var rows = 500;
6   var cols = 2000;
7
8   // Создаем две матрицы, которые будем перемножать
9   var m1 = new Matrix(rows, cols, MatrixFillType.Random);
10  var m2 = new Matrix(cols, rows, MatrixFillType.One);
11  // Начинаем замер времени
12  var sw = Stopwatch.StartNew();
```

```
13  var mProd = m1 * m2;
14  // Останавливаем замер времени выполнения операции
15  sw.Stop();
16  Console.WriteLine(sw.ElapsedMilliseconds);
17
18  namespace Parallel
19  {
20      /// <summary>
21      /// Класс матриц
22      /// </summary>
23      public class Matrix
24      {
25          // Генератор случайных чисел для заполнения матрицы
26          private static Random rnd = new Random();
27          // Массив для хранения элементов матрицы
28          private double[,] elems;
29
30          /// <summary>
31          /// Количество строк матрицы
32          /// </summary>
33          public int Rows => elems.GetLength(0);
34
35          /// <summary>
36          /// Количество столбцов матрицы
37          /// </summary>
38          public int Cols => elems.GetLength(1);
39
40          /// <summary>
41          /// Индексатор, позволяющий получить элемент матрицы по индексам
42          /// </summary>
43          /// <param name="row">Номер строки</param>
44          /// <param name="col">Номер столбца</param>
45          /// <returns>Элемент матрицы, расположенный в указанной позиции</returns>
46          public double this[int row, int col]
47          {
48              get => elems[row, col];
49              set => elems[row, col] = value;
50          }
51
52          /// <summary>
53          /// Вспомогательный конструктор, который позволяет задать массив элементов
54          /// </summary>
55          /// <param name="rows">Количество строк матрицы</param>
56          /// <param name="cols">Количество столбцов матрицы</param>
57          private Matrix(int rows, int cols)
58          {
59              elems = new double[rows, cols];
60          }
61
62          /// <summary>
```

```
63          /// Конструктор для создания матрицы указанного размера, элементы которой
64          /// задаются определенным образом
65          /// </summary>
66          /// <param name="rows">Число строк матрицы</param>
67          /// <param name="cols">Число столбцов матрицы</param>
68          /// <param name="fillType">Способ заполнения матрицы</param>
69          public Matrix(int rows, int cols, MatrixFillType fillType):
70                                                              this(rows, cols)
71          {
72              switch (fillType)
73              {
74                  case MatrixFillType.One:
75                      for (int i = 0; i < Math.Min(rows, cols); i++)
76                          elems[i, i] = 1;
77                      break;
78                  case MatrixFillType.Random:
79                      elems=Iterate(elems,(i,j)=>rnd.NextDouble()*20.0-10.0);
80                      break;
81              }
82          }
83
84          /// <summary>
85          /// Конструктор, позволяющий создать матрицу по заданному прямоугольному массиву.
86          /// </summary>
87          /// <param name="elems">Массив элементов матрицы</param>
88          public Matrix(double[,] elems) :
89                                  this(elems.GetLength(0), elems.GetLength(1))
90          {
91              Array.Copy(elems, this.elems, elems.Length);
92          }
93
94          /// <summary>
95          /// Последовательный способ перебора элементов
96          /// </summary>
97          /// <param name="elems">Массив элементов, которые необходимо задать</param>
98          /// <param name="calcElem">Функция заполнения элемента по индексам</param>
99          /// <returns>Заполненный массив элементов</returns>
100         private static double[,] Iterate(double[,] elems,
101                                          Func<int, int, double> calcElem)
102         {
103             for (int i = 0; i < elems.GetLength(0); i++)
104             {
105                 for (int j = 0; j < elems.GetLength(1); j++)
106                 {
107                     elems[i, j] = calcElem(i, j);
108                 }
109             }
110             return elems;
111         }
112
```

```
113             /// <summary>
114             /// Оператор умножения двух матриц.
115             /// </summary>
116             /// <param name="x">Левая матрица</param>
117             /// <param name="y">Правая матрица</param>
118             /// <returns>Новая матрица - результат произведения</returns>
119             /// <exception cref="Exception">Произведение нельзя вычислить,
120             /// когда число столбцов левой матрицы не совпадает с числом строк правой.
121             /// В этом случае генерируется исключение.
122             /// </exception>
123             public static Matrix operator *(Matrix x, Matrix y)
124             {
125                 if (x.Cols != y.Rows)
126                     throw new Exception("Матрицы имеют несовместимые размеры!");
127                 var res = new double[x.Rows, y.Cols];
128                 return new Matrix(Iterate(res, (i, j) =>
129                 {
130                     var e = 0.0;
131                     for (int k = 0; k < x.Cols; k++)
132                         e += x[i, k] * y[k, j];
133                     return e;
134                 }));
135             }
136         }
137 
138         /// <summary>
139         /// Способ заполнения матрицы значениями
140         /// </summary>
141         public enum MatrixFillType
142         {
143             /// <summary>
144             /// Матрица заполняется нулями
145             /// </summary>
146             Zero,
147             /// <summary>
148             /// Создает единичную матрицу
149             /// </summary>
150             One,
151             /// <summary>
152             /// Создает матрицу, заполненную случайными величинами
153             /// </summary>
154             Random
155         }
156     }
```

В приведенном примере основное внимание следует обратить на оператор умножения, а также метод `Iterate()`, который он вызывает для задания элементов матрицы.

Основная идея подхода состоит в том, что метод `Iterate()` будет перебирать все элементы полученного в параметре массива и заполнять их

значениями, которые будет генерировать передаваемая методу функция `calcElem`. Таким образом, `Iterate()` позволяет выделить универсальную часть перебора элементов матрицы, и может быть использован как при инициализации матрицы, так и при выполнении любых операций с ней. Именно этот метод мы будем модифицировать для добавления в приложение многопоточности. Это должно позволить повысить скорость вычислений.

Отметим, что время вычислений для заданных в программе размеров матриц составило порядка 13 секунд.

Изменим метод `Iterate()`, добавив в него потоки. Число потоков сделаем равным числу строк матрицы, таким образом, чтобы работа каждого потока заключалась в вычислении значений элементов каждой строки.

```
94   /// <summary>
95   /// Перебор элементов на потоках (число потоков соответствует числу строк)
96   /// </summary>
97   /// <param name="elems">Массив элементов, которые необходимо задать</param>
98   /// <param name="calcElem">Функция заполнения элемента по индексам</param>
99   /// <returns>Заполненный массив элементов</returns>
100  private static double[,] Iterate(double[,] elems,
101                                   Func<int, int, double> calcElem)
102  {
103      var threads = new List<Thread>();
104      // Каждый поток будет вычислять элементы одной строки
105      var parts = elems.GetLength(0);
106      // Создаем потоки
107      for (int t = 0; t < parts; t++)
108      {
109          threads.Add(new Thread((start) =>
110          {
111              // Откладываем запуск потоков на 300 мс. для того,
112              // чтобы основной поток успел запустить все вычислительные потоки
113              // до того, как они начнут работать.
114              Thread.Sleep(300);
115              if (start is int i)
116              {
117                  for (int j = 0; j < elems.GetLength(1); j++)
118                  {
119                      elems[i, j] = calcElem(i, j);
120                  }
121              }
122          }));
123      }
124      var cnt = 0;
125      // Запускаем потоки на выполнение
126      foreach (var t in threads) t.Start(cnt++);
127      // Дожидаемся завершения работы потоков
128      foreach (var t in threads) t.Join();
129      // Возвращаем полученный результат
```

```
130         return elems;
131     }
```

В этом случае время выполнения приложения значительно уменьшается до 1,6 секунды, однако в течение работы программы, вся система будет сильно «тормозить», поскольку создание большого количества потоков займет большую часть вычислительных ресурсов. Компьютером на время работы приложения будет сложно пользоваться.

Чтобы избавиться от негативных эффектов, возникающих при создании множества потоков, попробуем сократить их количество до значения, соответствующего числу процессоров (ядер процессора) в системе. Получить это значение можно благодаря статическому свойству `ProcessorCount` класса `Environment`. Данный вариант реализации описан в приведенном ниже фрагменте кода.

```
94  /// <summary>
95  /// Перебор элементов на потоках (число потоков = числу процессоров)
96  /// </summary>
97  /// <param name="elems">Массив элементов, которые необходимо задать</param>
98  /// <param name="calcElem">Функция заполнения элемента по индексам</param>
99  /// <returns>Заполненный массив элементов</returns>
100 private static double[,] Iterate(double[,] elems,
101                                  Func<int, int, double> calcElem)
102 {
103     Thread[] ts = new Thread[Environment.ProcessorCount];
104     for (int t = 0; t < Environment.ProcessorCount; t++)
105     {
106         ts[t] = new Thread((s) =>
107         {
108             for (int i = (int)s;
109                  i < elems.GetLength(0);
110                  i += Environment.ProcessorCount
111                 )
112             {
113                 for (int j = 0; j < elems.GetLength(1); j++)
114                     elems[i, j] = calcElem(i, j);
115             }
116         });
117         ts[t].Start(t);
118     }
119     foreach (var t in ts) t.Join();
120     return elems;
121 }
```

При использовании такого подхода время вычислений сократилось до 1,3 секунды. Однако, если вспомнить, что в предыдущем примере была введена задержка в 0,3с. перед стартом потоков, время самих вычислений можно считать не изменившимся. Таким образом, сохранив ту же самую

производительность, нам удалось добиться существенного сокращения числа потоков. При этом вся вычислительная система позволит пользователю общаться с ней в комфортном режиме.

Последний вариант является самым быстродействующим из рассмотренных и не производит негативных эффектов. Однако и у него есть недостатки.

При создании параллельного алгоритма зачастую требуется осуществлять балансировку нагрузки, то есть распределение процесса выполнения заданий между процессорами с целью оптимизации использования ресурсов и сокращения времени вычисления.

Рассмотрим ситуацию, когда разным потокам из последнего примера нужно будет решать разные по объему задачи. Это может привести к ситуации, когда некоторые потоки завершатся раньше остальных. В итоге часть времени, затраченного на вычисления, процессор будет использоваться неэффективно: например, в течение некоторого периода может работать 1-2 потока, которым достались самые трудоемкие части.

Изменим оператор умножения, чтобы смоделировать такую ситуацию.

---

ОТМЕТИМ, ЧТО ПОСЛЕ ВНЕСЕНИЯ ТАКИХ ИЗМЕНЕНИЙ, БУДЕТ ВЫЧИСЛЯТЬСЯ УЖЕ НЕ ПРОИЗВЕДЕНИЕ ДВУХ МАТРИЦ, А КАКОЕ-ТО ДРУГОЕ ЗНАЧЕНИЕ, НО ДЛЯ НАШЕГО ПРИМЕРА ЭТО НЕ ПРИНЦИПИАЛЬНО.

---

```
113  /// <summary>
114  /// Оператор выполнения некоторого действия над двумя матрицами.
115  /// </summary>
116  /// <param name="x">Левая матрица</param>
117  /// <param name="y">Правая матрица</param>
118  /// <returns>Новая матрица - результат произведения</returns>
119  /// <exception cref="Exception">Операцию нельзя выполнить,
120  /// когда число столбцов левой матрицы не совпадает с числом строк правой.
121  /// В этом случае генерируется исключение.
122  /// </exception>
123  public static Matrix operator *(Matrix x, Matrix y)
124  {
125      if (x.Cols != y.Rows)
126          throw new Exception("Матрицы имеют несовместимые размеры!");
127      var res = new double[x.Rows, y.Cols];
128      return new Matrix(Iterate(res, (i, j) =>
129      {
130          var e = 0.0;
131          var pc = Environment.ProcessorCount;
132          for (int k = 0; k < x.Cols; k++)
133          {
134              var xe = x[i, k];
135              for (int l = 0; l < (i % pc) * (i % pc); l++)
```

```
136             {
137                 xe += x[i, k];
138             }
139             e += xe * y[k, j];
140         }
141         return e;
142     }));
143 }
```

Здесь при вычислении значения элемента был добавлен цикл, число итераций которого пропорционально остатку от деления номера строки на число процессоров в системе. В то же время, каждый поток в методе `Iterate()` в последнем примере, как раз проходит по строкам матрицы с шагом равным числу процессоров. Таким образом, в части случаев добавленный цикл может всегда иметь небольшое число повторений, а в других – близким к максимальному. В связи с этим, время выполнения одних потоков будет существенно больше, чем других.

Общее время выполнения приложения возрастет до 70,5 секунд. На это влияют две составляющие: само добавление дополнительных действий, а также появившаяся неравномерность и неэффективное использование процессорного времени. Попробуем оптимизировать код для устранения указанной неэффективности.

Решить подобную проблему может помочь разбиение всего решения на более мелкие фрагменты, как это было сделано во втором примере. (При этом подходе время вычислений может сократиться до 43,3 секунд.) Однако, как уже отмечалось ранее, в этом случае система начнет работать крайне медленно и пользователь не сможет полноценно использовать компьютер во время работы нашей программы.

В описанной ситуации оптимальным решением будет задействование задач и класса `Task`. Его применение позволит использовать более мелкое разбиение, но за счет встроенного в библиотеку TPL механизма планирования задач и их назначения на ограниченное число потоков из пула, не перегрузить всю систему.

```
94  /// <summary>
95  /// Метод перебора элементов матрицы на базе задач
96  /// </summary>
97  /// <param name="elems">Массив элементов, которые нужно перебрать,
98  /// задав им новые значения
99  /// </param>
100 /// <param name="calcElem">Функция, которая по указанным индексам
101 /// генерирует значение элемента матрицы
102 /// </param>
103 /// <returns>Массив измененных элементов матрицы</returns>
104 private static double[,] Iterate(double[,] elems,
```

```
105                                         Func<int, int, double> calcElem)
106 {
107     var parts = elems.GetLength(0);
108     var tasks = new List<Task>();
109     for (int t = 0; t < parts; t++)
110     {
111         var tsk = Task.Factory.StartNew((start) =>
112         {
113             if (start is int i)
114             {
115                 for (int j = 0; j < elems.GetLength(1); j++)
116                 {
117                     elems[i, j] = calcElem(i, j);
118                 }
119             }
120         }, t);
121         tasks.Add(tsk);
122     }
123     Task.WaitAll(tasks.ToArray());
124     return elems;
125 }
```

Этот вариант, обеспечивая высокую скорость вычислений в течение порядка 42,6 секунд (примерно как и в случае множества потоков), дополнительно позволяет сохранить отзывчивость системы, что демонстрирует преимущества применения библиотеки TPL.

### Класс Parallel

Еще одним классом библиотеки TPL является класс `Parallel`, который позволяет в некоторых случаях упростить запись некоторых распространенных конструкций, например, циклов.

Этот класс содержит ряд статических методов для реализации параллельно выполняющихся циклов или параллельно выполняющихся независимых задач. Рассмотрим некоторые из них подробнее.

**Таблица II-6**

**Класс Parallel**

| Методы | Описания |
|---|---|
| `static ParallelLoopResult For (int fromInclusive, int toExclusive, ParallelOptions` | Выполняет цикл for, обеспечивая возможность параллельного выполнения итераций и настройки параметров цикла. Рассмотрим параметры метода.<br>*fromInclusive:* |

| Методы | Описания |
|---|---|
| `options, Action<int, ParallelLoopState> body);` | начальный индекс (включительно);<br>*toExclusive:*<br>конечный индекс, не включительно;<br>*options:*<br>объект, используемый для настройки поведения этой операции;<br>*body:*<br>делегат, вызываемый один раз за итерацию.<br>Наиболее интересным для нас в классе `ParallelOptions` является свойство `MaxDegreeOfParallelism`, позволяющее задавать максимальную степень параллелизма (число одновременно выполняющихся потоков). |
| `static ParallelLoopResult ForEach<TSource, TLocal> (IEnumerable <TSource> source, ParallelOptions parallelOptions, Func<TLocal> localInit, Func<TSource, ParallelLoopState, TLocal, TLocal> body, Action<TLocal> localFinally);` | Выполняет операцию `foreach`, обеспечивая возможность параллельного выполнения итераций и настройки параметров цикла.<br>`TSource` – тип данных в источнике.<br>Параметры метода:<br>*source:*<br>перечислимый источник данных.<br>*parallelOptions:*<br>объект, используемый для настройки поведения этой операции.<br>*body:*<br>делегат, вызываемый один раз за итерацию. |
| `static void Invoke( ParallelOptions parallelOptions,` | Этот метод может использоваться для потенциально параллельного выполнения набора операций. |

| Методы | Описания |
|---|---|
| `array<Action> actions)` | Этот метод не осуществляет возврат до завершения каждой из операций. |

ОТМЕТИМ ТАКЖЕ, ЧТО КАЖДАЯ ИТЕРАЦИЯ ПАРАЛЛЕЛЬНОГО ЦИКЛА, КАК И ВЫПОЛНЕНИЕ ДЕЙСТВИЙ В МЕТОДЕ `INVOKE()` ВЫПОЛНЯЕТСЯ НА БАЗЕ ЗАДАЧ.

Попробуем переписать код последнего примера с применением параллельного цикла.

```
1   /// <summary>
2   /// Метод перебора элементов матрицы на базе параллельного цикла
3   /// </summary>
4   /// <param name="elems">Массив элементов, которые нужно перебрать,
5   /// задав им новые значения
6   /// </param>
7   /// <param name="calcElem">Функция, которая по указанным индексам
8   /// генерирует значение элемента матрицы
9   /// </param>
10  /// <returns>Массив измененных элементов матрицы</returns>
11  private static double[,] Iterate(double[,] elems,
12                                   Func<int, int, double> calcElem)
13  {
14      var parts = elems.GetLength(0);
15      var tasks = new List<Task>();
16      // Задаем максимально допустимое число параллельно используемых потоков
17      var po = new ParallelOptions {
18          MaxDegreeOfParallelism = Environment.ProcessorCount
19      };
20      // Запускаем параллельный цикл
21      System.Threading.Tasks.Parallel.For(0, parts, po, (i) =>
22      {
23          for (int j = 0; j < elems.GetLength(1); j++)
24          {
25              elems[i, j] = calcElem(i, j);
26          }
27      });
28      return elems;
29  }
```

Такой вариант реализации позволил выполнить операцию за 40,4 секунды.

## ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

1. Написать классы для вычисления интегралов при помощи приближенных методов (левых прямоугольников, средних прямоугольников, трапеции, Симпсона) с использованием последовательных и параллельных алгоритмов. Сравнить время вычисления действий при использовании этих классов.
2. Написать класс для поиска всех простых чисел в заданном диапазоне и получения упорядоченного по возрастанию результата в файле. Реализовать в классе методы, использующие последовательный и параллельный алгоритм поиска. Определить время решения задачи в каждом случае. При реализации параллельных алгоритмов использовать задачи (`Task`).
3. Написать 2 класса, для работы с матрицами в режиме последовательной и параллельной обработки. Реализовать операции умножения матрицы на число, сложения двух матриц, умножения матриц. Сравнить время вычислений при использовании этих классов. При реализации параллельных алгоритмов использовать возможности класса `Parallel`.
4. Написать классы для осуществления сортировки массивов различными методами применяя последовательные и параллельные варианты одинаковых алгоритмов. Сравнить время их выполнения.
5. Написать систему классов для реализации работы многопоточной модели «Поставщик-Потребитель» с несколькими поставщиками и одним потребителем, используя для этих целей методы приостановки потоков из класса `Monitor`.
6. Написать класс блокирующей очереди лимитированного размера. Такой класс должен позволять считывающим потокам получать данные из очереди до тех пор, пока они там есть, а записывающим потокам добавлять данные в очередь, пока та будет содержать не более N элементов. При нарушении условий, потоки должны блокироваться до момента наступления возможности чтения/записи.

# СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ


1. Хорев, П. Б. Объектно-ориентированное программирование с примерами на C# : учебное пособие / П.Б. Хорев. – Москва : ФОРУМ : ИНФРА-М, 2020. – 200 с. – (Высшее образование: Бакалавриат). – ISBN 978-5-00091-680-3. – Текст : электронный. – URL: https://znanium.com/catalog/product/1069921 (дата обращения: 22.10.2023). – Режим доступа: по подписке.
2. Гуриков, С. Р. Введение в программирование на языке Visual C# : учебное пособие / С.Р. Гуриков. – Москва : ФОРУМ : ИНФРА-М, 2020. – 447 с. – (Высшее образование: Бакалавриат). – ISBN 978-5-00091-458-8. – Текст : электронный. – URL: https://znanium.com/catalog/product/1092167 (дата обращения: 22.10.2023). – Режим доступа: по подписке.
3. Лисьев, Г. А. Программное обеспечение компьютерных сетей и web-серверов : учебное пособие / Г.А. Лисьев, П.Ю. Романов, Ю.И. Аскерко. – Москва : ИНФРА-М, 2023. – 145 с. – (Высшее образование: Бакалавриат). – DOI 10.12737/textbook_5a93ba6860adc5.11807424. – ISBN 978-5-16-013565-6. – Текст : электронный. – URL: https://znanium.com/catalog/product/1914008 (дата обращения: 22.10.2023). – Режим доступа: по подписке.
4. Медведев, М. А. Программирование на СИ#: Учебное пособие / Медведев М.А., Медведев А.Н., – 2-е изд., стер. – Москва :Флинта, Изд-во Урал. ун-та, 2017. – 64 с. ISBN 978-5-9765-3169-7. – Текст : электронный. – URL: https://znanium.com/catalog/product/948428 (дата обращения: 22.10.2023). – Режим доступа: по подписке.
5. Сердюк, Ю. П. Параллельное программирование для многоядерных процессоров / Сердюк Ю. П. , Петров А. В. – Москва : Национальный Открытый Университет "ИНТУИТ", 2016. – Текст : электронный // ЭБС "Консультант студента" : [сайт]. – URL : https://www.studentlibrary.ru/book/intuit_239.html (дата обращения: 22.10.2023). – Режим доступа : по подписке.
6. Маклецов С. В. Курс «Объектно-ориентированное программирование». URL: https://edu.kpfu.ru/course/view.php?id=4710 (дата обращения: 22.10.2023). – Режим доступа: по подписке.
7. Полное руководство по языку программирования C# 11 и платформе .NET 7. [Электронный ресурс] / Metanit.com. – URL: https://metanit.com/sharp/tutorial/ (дата обращения: 22.10.2023). – Режим доступа: свободный.

*Сергей Владиславович Маклецов*

# Объектно-ориентированное программирование на языке C#

Учебное пособие