

ГЛАВА 18. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

Оглавление

§18.1 Многопоточное программирование	1
§18.1.1 Определение потоков.....	1
§18.1.2 Пример многопоточности	2
§18.1.3 Процессы	3
§18.1.4 Классификация потоков в среде .NET Framework	3
§18.1.5 Цели применения нескольких потоков	3
§18.2 Реализация потоков.....	4
§18.2.1 Класс Thread	4
§18.2.2 Создание потоков	8
§18.2.3 Примеры работы потоков.....	10
§18.3 Потоки с параметрами	13

§18.1 Многопоточное программирование

§18.1.1 Определение потоков

Одним из ключевых аспектов в современном программировании является **многопоточность**. Многопоточность позволяет увеличивать скорость реагирования приложения и, если приложение работает в многопроцессорной или многоядерной системе, его пропускную способность.

Ключевым понятием при работе с многопоточностью является **поток**. **Поток** - основная единица исполнения в операционной системе Windows. Все программы в этой ОС выполняются в потоках. Для того чтобы выполнить код какой-либо программы, ОС сначала создает поток, выделяет для него память, загружает в нее код программы и только после этого начинается выполнение.

Если говорить простым языком, то поток – это некая независимая последовательность инструкций для выполнения того или иного действия в программе. В одном конкретном потоке выполняется одна конкретная последовательность действий.

Совокупность таких потоков, выполняемых в программе параллельно называется многопоточностью программы.

§18.1.2 Пример многопоточности

Для лучшего понимания многопоточности можно представить **следующий пример**. Допустим, что наша программа и данные, которые в ней содержатся – это офис с различными предметами (папками, столами, стульями, ручками), а потоки – это работники данного офиса (изначально у нас только один работник, выполняющий всю работу), и каждый работник занимается теми делами, которые ему было сказано выполнять. Работники могут выполнять одинаковые задания, а могут и различные. В случае выполнения какой-либо одной задачи, несколько работников справятся быстрее, чем один.

Например, если один работник будет собирать шкаф час, то вдвоём они могут управиться уже за полчаса. Однако не стоит переусердствовать в количестве работников (потоков). Математически, если нанять 4 работника, то шкаф соберется за 15 минут, если нанять 60 работников – за 1 минуту, а если нанять 3600, то вообще за секунду, но ведь на деле это неверно. Работники будут только мешать друг другу, толкаться, отнимать друг у друга детали, и процесс сборки шкафа может затянуться очень надолго.

Так же и с потоками. Чем больше потоков, тем выше вероятность, что они будут мешать друг другу выполнять свою работу. Например, если заставить работать огромное количество потоков с одними и теми же данными, потокам придётся выстраиваться в очередь для их обработки (например, если тем же 3600 рабочим дать какое-либо письменное задание, но предоставить им для этого дела всего одну ручку, то работникам, естественно, придётся становиться друг за другом в очередь за ручкой, чтобы после её получения выполнить поставленную задачу. Времени это займёт довольно много).

Потоки надо распределять с умом и исключительно в случаях, когда это действительно необходимо для ускорения работы программы либо для повышения производительности.

§18.1.3 Процессы

Потоки в Windows принадлежат **процессам**. **Процесс** — это исполнение программы. Операционная система использует процессы для разделения исполняемых приложений.

Программа на C# запускается в единственном потоке, однако процессу может принадлежать множество потоков. Эти потоки могут выполняться одновременно на многопроцессорных системах, что значительно увеличивает производительность программы. В однопроцессорных системах потоки выполняются последовательно, получая по очереди кванты времени (20 мс), однако даже такое выполнение в некоторых случаях приводит к увеличению производительности программы.

§18.1.4 Классификация потоков в среде .NET Framework

Язык C# имеет встроенную поддержку многопоточности, а среда .NET Framework предоставляет сразу несколько классов для работы с потоками, что в совокупности помогает реализовывать и настраивать многопоточность в проектах.

В среде .NET Framework существует два типа потоков: **основной** и **фоновый** (вспомогательный). Фоновые потоки отличаются от основных только в одном аспекте: если первым завершится основной поток, то фоновые потоки в его процессе будут также принудительно остановлены, если же первым завершится фоновый поток, то это не повлияет на остановку основного потока — тот будет продолжать функционировать до тех пор, пока не выполнит всю работу и самостоятельно не остановится.

§18.1.5 Цели применения нескольких потоков

Использовать несколько потоков необходимо, чтобы увеличить скорость реагирования приложения и воспользоваться преимуществами

многопроцессорной или многоядерной системы, чтобы увеличить пропускную способность приложения.

Представьте себе классическое приложение, в котором основной поток отвечает за элементы пользовательского интерфейса и реагирует на действия пользователя. Используйте рабочие потоки для выполнения длительных операций, которые, в противном случае будут занимать основной поток, в результате чего пользовательский интерфейс будет недоступен. Для более оперативной реакции на входящие сообщения или события также можно использовать выделенный поток связи с сетью или устройством.

Если программа выполняет операции, которые могут выполняться параллельно, можно уменьшить общее время выполнения путем выполнения этих операций в отдельных потоках и запуска программы в многопроцессорной или многоядерной системе. В такой системе использование многопоточности может увеличить пропускную способность, а также повысить скорость реагирования.

§18.2 Реализация потоков

§18.2.1 Класс Thread

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен `System.Threading`. В нем определен класс, представляющий отдельный поток - класс `Thread`.

Класс `Thread` определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем.

Основные свойства класса `Thread` описаны в таблице 18.1.

Таблица 18.1 – Свойства класса Thread

Наименование свойства	Описание
<code>CurrentContext</code>	Статическое свойство, которое позволяет получить контекст, в котором выполняется поток

Наименование свойства	Описание
CurrentThread	Статическое свойство, которое возвращает ссылку на выполняемый поток
IsAlive	Свойство, которое указывает, работает ли поток в текущий момент
IsBackground	Свойство, которое указывает, является ли поток фоновым
Name	Свойство, которое содержит имя потока
Priority	Свойство, которое хранит приоритет потока ¹
ThreadState	Свойство, которое возвращает состояние потока ²

Некоторые из методов класса Thread описаны в таблице 18.2.

Таблица 18.2 – Методы класса Thread

Наименование метода	Описание
Sleep	Статический метод, который останавливает поток на определенное количество миллисекунд ³
Abort	Метод, который уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки

¹ <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.threadpriority?view=net-5.0>

² <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.threadstate?view=net-5.0>

³ <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.thread.sleep?view=net-5.0>

Наименование метода	Описание
	завершенности потока следует опрашивать его свойство ThreadState ⁴
Interrupt	Метод, который прерывает поток на некоторое время ⁵
Join	Метод блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод ⁶
Start	Метод запускает поток ⁷

Используем вышеописанные свойства и методы для получения информации о потоке:

Листинг 18.1 – Получение информации о потоке

1	using System.Threading;
2	using System.Windows.Forms;
3	namespace WindowsFormsApp13
4	{
5	public partial class Form1 : Form
6	{
7	public Form1()
8	{
9	InitializeComponent();
10	Thread t = Thread.CurrentThread;
11	t.Name = "Метод Main";
12	richTextBoxThreadInfo.Text = \$"Имя потока: {t.Name}" +
	"\n" +
13	\$"Запущен ли поток: {t.IsAlive}" + "\n" +
14	\$"Приоритет потока: {t.Priority}" + "\n" +
15	\$"Статус потока: {t.ThreadState}" + "\n" +
16	\$"Домен приложения: {Thread.GetDomain().FriendlyName}";

⁴ <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.thread.abort?view=net-5.0>

⁵ <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.thread.interrupt?view=net-5.0>

⁶ <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.thread.join?view=net-5.0>

⁷ <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.thread.start?view=net-5.0>

17	}
18	}
19	}

В этом случае мы получим примерно следующий вывод:

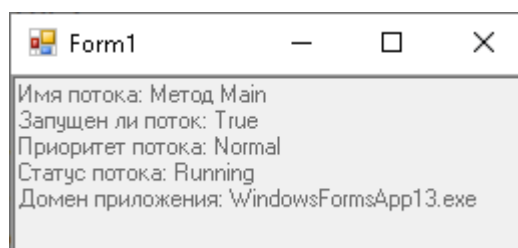


Рисунок 18.1 – Вывод состояния потока

Статусы потока содержатся в перечислении ThreadState:

Таблица 18.3 – Статусы потока

Статус потока	Описание статуса
Aborted	поток остановлен, но пока еще окончательно не завершен
AbortRequested	для потока вызван метод Abort, но остановка потока еще не произошла
Background	поток выполняется в фоновом режиме
Running	поток запущен и работает (не приостановлен)
Stopped	поток завершен
StopRequested	поток получил запрос на остановку
Suspended	поток приостановлен
SuspendRequested	поток получил запрос на приостановку
Unstarted	поток еще не был запущен
WaitSleepJoin	поток заблокирован в результате действия методов Sleep или Join

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение **Unstarted**. Запустив поток, мы изменим его статус на **Running**. Вызвав метод **Sleep**, статус изменится на **WaitSleepJoin**. А применяя метод **Abort**, мы тем самым переведем поток в состояние

`AbortRequested`, а затем `Aborted`, после чего поток окончательно завершится.

Когда в программе фигурирует несколько потоков, выбор процессором следующего потока для выполнения не является случайным. Дело в том, что у каждого потока имеется значение приоритета, чем выше приоритет потока, тем важнее для процессора предоставить время и ресурсы для выполнения именно ему. Если же приоритет потока не слишком высокий, значит он может и подождать в очереди, пока выполняются более приоритетные потоки.

Приоритеты потоков располагаются в перечислении `ThreadPriority`:

- `Lowest` – самый низкий
- `BelowNormal` – ниже среднего
- `Normal` – стандартный
- `AboveNormal` – выше среднего
- `Highest` – самый высокий

По умолчанию потоку задается значение `Normal`. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет `Highest`. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

§18.2.2 Создание потоков

Используя класс `Thread`, мы можем выделить в приложении несколько потоков, которые будут выполняться одновременно.

Во-первых, для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

Для создания нового потока используется делегат `ThreadStart`, который получает в качестве параметра выполняемый метод. И чтобы запустить поток, вызывается метод `Start`.

Рассмотрим пример:

1	<code>using System.Threading;</code>
2	<code>class Program</code>
3	<code>{</code>
4	<code> static void Main(string[] args)</code>
5	<code> {</code>
6	<code> // создаем новый поток</code>
7	<code> Thread myThread = new Thread(new</code> <code>ThreadStart(Count));</code>
8	<code> myThread.Start(); // запускаем поток</code>
9	<code> for (int i = 1; i < 9; i++)</code>
10	<code> {</code>
11	<code> Console.WriteLine("Главный поток:");</code>
12	<code> Console.WriteLine(i * i);</code>
13	<code> Thread.Sleep(300);</code>
14	<code> }</code>
15	<code> Console.ReadLine();</code>
16	<code> }</code>
17	<code> public static void Count()</code>
18	<code> {</code>
19	<code> for (int i = 1; i < 9; i++)</code>
20	<code> {</code>
21	<code> Console.WriteLine("Второй поток:");</code>
22	<code> Console.WriteLine(i + i);</code>
23	<code> Thread.Sleep(400);</code>
24	<code> }</code>
25	<code> }</code>
26	<code>}</code>

Здесь новый поток будет производить действия, определенные в методе `Count`. В данном случае это возведение в квадрат числа и вывод его на экран. И после каждого умножения с помощью метода `Thread.Sleep` мы усыпляем поток на 400 миллисекунд.

Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока:

```
Thread myThread = new Thread(new ThreadStart(Count));
```

В конструктор передается делегат `ThreadStart`, который в качестве параметра принимает метод `Count`. И следующей строкой

```
myThread.Start()
```

мы запускаем поток. После этого управление передается главному потоку, и выполняются все остальные действия, определенные в методе `Main`.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом `Main`, и второй поток. Кроме действий по созданию второго потока, в главном потоке также производятся некоторые вычисления. Как только все потоки отработают, программа завершит свое выполнение.

Подобным образом мы можем создать и три, и четыре, и целый набор новых потоков, которые смогут решать те или иные задачи.

Существует еще одна форма создания потока:

```
Thread myThread = new Thread(Count);
```

Хотя в данном случае явным образом мы не используем делегат `ThreadStart`, но неявно он создается. Компилятор `C#` выводит делегат из сигнатуры метода `Count` и вызывает соответствующий конструктор.

§18.2.3 Примеры работы потоков

Пример 1

На примере посмотрим, на что влияет приоритетность потоков в `C#`. Мы решили взять для примера программу с тремя потоками, каждый из которых будет выводить в консоль цифры от 0 до 9, от 10 до 19 и от 20 до 29 соответственно. Поставим перед собой задачу вывести в консоль все эти числа последовательно от 0 до 29.

Если мы пренебрежем приоритетами при постановке данной задачи, то у нас никак не получится вывести все наши числа по очереди. Так как у всех трёх потоков одинаковый приоритет, процессору, по сути, будет всё равно, какой за каким потоки выводить, и у нас частенько будет выходить плохо отсортированный набор чисел.

Попробуем реализовать всё так, чтобы у нас получился необходимый нам результат.

Листинг 18.2 – Изменение приоритета потоков

1	static void mythread1()
2	{
3	for (int i = 0; i < 10; i++)
4	{
5	Console.WriteLine("Поток 1 выводит " + i);
6	}
7	}
8	static void mythread2()
9	{
10	for (int i = 10; i < 20; i++)
11	{
12	Console.WriteLine("Поток 2 выводит " + i);
13	}
14	}
15	static void mythread3()
16	{
17	for (int i = 20; i < 30; i++)
18	{
19	Console.WriteLine("Поток 3 выводит " + i);
20	}
21	}
22	static void Main(string[] args)
23	{
24	Thread thread1 = new Thread(mythread1);
25	Thread thread2 = new Thread(mythread2);
26	Thread thread3 = new Thread(mythread3);
27	thread1.Priority = ThreadPriority.Highest;
28	thread2.Priority = ThreadPriority.AboveNormal;
29	thread3.Priority = ThreadPriority.Lowest;
30	thread1.Start();
31	thread2.Start();
32	thread3.Start();
33	Console.ReadLine();
34	}
35	}

У нас имеется три потока и три метода, в которых они выполняются. Первый метод выводит на экран числа от 0 до 9, второй – от 10 до 19, третий – с 20 по 29.

В методе `Main` мы задаём приоритеты потокам. Так как нам надо сначала вывести числа из первого потока, мы устанавливаем ему самый высокий

приоритет. Следующий за ним второй поток имеет приоритет чуть ниже, поэтому будет выполняться вторым, третий поток имеет самый низкий приоритет, поэтому будет выполняться после всех остальных приоритетов.

Результат работы программы представлен на рисунке 18.2.



```
Поток 1 выводит 0
Поток 1 выводит 1
Поток 1 выводит 2
Поток 1 выводит 3
Поток 1 выводит 4
Поток 1 выводит 5
Поток 1 выводит 6
Поток 1 выводит 7
Поток 1 выводит 8
Поток 1 выводит 9
Поток 2 выводит 10
Поток 2 выводит 11
Поток 2 выводит 12
Поток 2 выводит 13
Поток 2 выводит 14
Поток 2 выводит 15
Поток 2 выводит 16
Поток 2 выводит 17
Поток 2 выводит 18
Поток 2 выводит 19
Поток 3 выводит 20
Поток 3 выводит 21
Поток 3 выводит 22
Поток 3 выводит 23
Поток 3 выводит 24
Поток 3 выводит 25
Поток 3 выводит 26
Поток 3 выводит 27
Поток 3 выводит 28
Поток 3 выводит 29
```

Рисунок 18.2

Наши числа вывелись в правильном порядке, и всё благодаря приоритетам.

Пример 2

Потоки в C# бывают приоритетными и фоновыми. Разница между ними в том, что если основной поток будет завершен, то и вложенные в него фоновые потоки также будут завершены принудительно. Если же фоновый поток завершится раньше основного, то на основной поток это не повлияет, и он продолжит свою работу.

Мы также описывали выше, как поменять тип потока в программе, как по умолчанию все потоки создаются приоритетными. Рассмотрим на практике, чем отличаются фоновые потоки в C# от основных.

Создадим программу, которая проливает свет на влияние типов потоков:

Листинг 18.3 – Изменение типа потока

1	<code>static void mythread1()</code>
2	<code>{</code>
3	<code>for (int i = 0; i < 1000000; i++)</code>
4	<code>{</code>
5	<code>Console.WriteLine("Поток 1 выводит " + i);</code>
6	<code>}</code>
7	<code>}</code>
8	<code>static void Main(string[] args)</code>
9	<code>{</code>
10	<code>Thread thread1 = new Thread(mythread1);</code>
11	<code>thread1.IsBackground = true;</code>
12	<code>thread1.Start();</code>
13	<code>Thread.Sleep(100);</code>
14	<code>}</code>
15	<code>}</code>

Здесь у нас имеется два потока – `thread1` и поток метода `Main`. Изначально они являются приоритетными, следовательно, работают независимо друг от друга, и, пока не закончится выполняться один поток, второй поток нельзя будет закончить принудительно. Однако в нашей программе мы присвоили потоку `thread1` другой тип – фоновый, в строке 11.

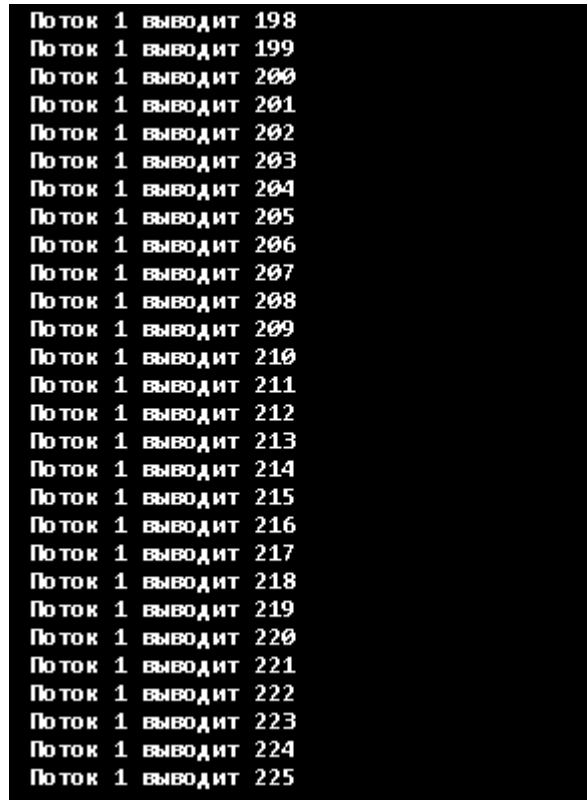
А так как данный поток вызывается в методе `Main`, значит он будет полностью зависеть от потока в этом методе.

Смысл нашей программы в следующем: поток `thread1` вызывается из метода `Main` и начинает работу в методе `mythread1`, который должен выводить на экран числа от 0 до 999999. Однако загвоздка в том, что практически после старта работа метода `Main` прекращается. Поток `thread1` никак не успеет вывести все 999999 чисел за столь короткий промежуток времени, но так как в нашей программе он является фоновым, то принудительно завершится вместе с завершением потока в методе `Main`.

Мы также добавили строку 13, который приостанавливает работу потока на время, указанное скобках (миллисекунды). В нашем случае мы приостанавливаем приоритетный поток исключительно для того, чтобы

успеть сделать скриншот вывода (тоже нелегкое дело, успеть сфотографировать экран за одну десятую секунды, но если сделать остановку значительно дольше, то поток `thread1` успеет вывести все числа).

Итак, после запуска программы и выжидания 100 мс у нас в консоли следующее:



```
Поток 1 выводит 198
Поток 1 выводит 199
Поток 1 выводит 200
Поток 1 выводит 201
Поток 1 выводит 202
Поток 1 выводит 203
Поток 1 выводит 204
Поток 1 выводит 205
Поток 1 выводит 206
Поток 1 выводит 207
Поток 1 выводит 208
Поток 1 выводит 209
Поток 1 выводит 210
Поток 1 выводит 211
Поток 1 выводит 212
Поток 1 выводит 213
Поток 1 выводит 214
Поток 1 выводит 215
Поток 1 выводит 216
Поток 1 выводит 217
Поток 1 выводит 218
Поток 1 выводит 219
Поток 1 выводит 220
Поток 1 выводит 221
Поток 1 выводит 222
Поток 1 выводит 223
Поток 1 выводит 224
Поток 1 выводит 225
```

Рисунок 18.3

Это последние числа, которые успел вывести поток `thread1` перед завершением выполнения потока в `Main`, затем консоль закрывается, и приложение завершает свою работу.

Для дополнительного примера можно убрать или заменить строку `thread1.IsBackground = true;` (тогда данный поток снова станет приоритетным), удалить `Thread.Sleep(100)`, добавить в конец метода `Main` строку `Console.ReadLine()` и снова запустить данную программу. Перед вами начнут проплывать огромное количество чисел, но вы не сможете остановить это действие посредством работы потока в `Main` (иными словами,

нажатием любой кнопки), пока поток `thread1` не выполнится полностью и не выведет все 999999 чисел.

§18.3 Потоки с параметрами

Для передачи параметров в поток используется делегат `ParameterizedThreadStart`. Его действие похоже на функциональность делегата `ThreadStart`. Рассмотрим на примере:

Листинг 18.4

1	<code>class Program</code>
2	<code>{</code>
3	<code>static void Main(string[] args)</code>
4	<code>{</code>
5	<code>int number = 4;</code>
6	<code>// создаем новый поток</code>
7	<code>Thread myThread = new Thread(new</code>
8	<code>ParameterizedThreadStart(Count));</code>
9	<code>myThread.Start(number);</code>
10	<code>for (int i = 1; i < 9; i++)</code>
11	<code>{</code>
12	<code>Console.WriteLine("Главный поток:");</code>
13	<code>Console.WriteLine(i * i);</code>
14	<code>Thread.Sleep(300);</code>
15	<code>}</code>
16	<code>Console.ReadLine();</code>
17	<code>}</code>
18	<code>public static void Count(object x)</code>
19	<code>{</code>
20	<code>for (int i = 1; i < 9; i++)</code>
21	<code>{</code>
22	<code>int n = (int)x;</code>
23	<code>Console.WriteLine("Второй поток:");</code>
24	<code>Console.WriteLine(i*n);</code>
25	<code>Thread.Sleep(400);</code>
26	<code>}</code>
27	<code>}</code>

После создания потока мы передаем метод `myThread.Start(number);` переменную, значение которой хотим передать в поток.

При использовании `ParameterizedThreadStart` мы сталкиваемся с ограничением: мы можем запускать во втором потоке только такой метод, который в качестве единственного параметра принимает объект типа `object`. Поэтому в данном случае нам надо дополнительно привести переданное значение к типу `int`, чтобы его использовать в вычислениях.

Но что делать, если нам надо передать не один, а несколько параметров различного типа? В этом случае на помощь приходит классовый подход:

Листинг 18.4

1	<code>class Program</code>
2	<code>{</code>
3	<code>static void Main(string[] args)</code>
4	<code>{</code>
5	<code>Counter counter = new Counter();</code>
6	<code>counter.x = 4;</code>
7	<code>counter.y = 5;</code>
8	<code>Thread myThread = new Thread(new</code>
	<code>ParameterizedThreadStart(Count));</code>
9	<code>myThread.Start(counter);</code>
10	<code>//.....</code>
11	<code>}</code>
12	<code>public static void Count(object obj)</code>
13	<code>{</code>
14	<code>for (int i = 1; i < 9; i++)</code>
15	<code>{</code>
16	<code>Counter c = (Counter)obj;</code>
17	<code>Console.WriteLine("Второй поток:");</code>
18	<code>Console.WriteLine(i*c.x *c.y);</code>
19	<code>}</code>
20	<code>}</code>
21	<code>}</code>
22	<code>public class Counter</code>
23	<code>{</code>
24	<code>public int x;</code>
25	<code>public int y;</code>
26	<code>}</code>

Сначала определяем специальный класс `Counter`, объект которого будет передаваться во второй поток, а в методе `Main` передаем его во второй поток.