**Лабораторная. Высокоуровневый язык параллельного программирования MC#**

**Задание 1**



Задание 2

Дана последовательность натуральных чисел {*a0…an–1*}. Создать приложение для поиска всех *ai*, являющихся простыми числами.

Язык параллельного программирования MC# предназначен для написания программ, работающих на всём спектре параллельных архитектур - от многоядерных процессоров до Grid-сетей. Единственное требование к таким системам со стороны MC# - на них должна быть установлена среда исполнения CLR (Common Language Runtime) с соответствующим набором библиотек. На машинах с операционной системой Windows реализацией такой среды является Microsoft .NET Framework

Язык MC# является адаптацией и развитием базовых идей языка Polyphonic C# на случай параллельных и распределенных вычислений. Язык Polyphonic C# был разработан в 2002г. в Microsoft Research Laboratory (г. Кембридж, Великобритания) Н. Бентоном (N. Benton), Л. Карделли (L. Cardelli) и Ц. Фурнье (C. Fournet). Целью его создания было добавление высокоуровневых средств асинхронного параллельного программирования в язык C# для использования в серверных и клиент-серверных приложениях на базе Microsoft .NET Framework.

Ключевая особенность языка Polyphonic C# заключается в добавлении к обычным, синхронным методам, так называемых "асинхронных" методов, которые предназначены играть в (многопоточных) программах две основные роли:

1. автономных методов, предназначенных для выполнения базовой вычислительной работы, и исполняемых в отдельных потоках, и
2. методов, предназначенных для доставки данных (сигналов) обычным, синхронным методам.

Для синхронизации нескольких асинхронных методов, а также асинхронных и синхронных методов, в язык C#, кроме того, были введены новые конструкции, получившие название связок (chords).

При этом исполнение Polyphonic C#-программ, по замыслу авторов этого языка, по-прежнему, предполагалось либо на одной машине, либо на нескольких машинах, с зафиксированными на них асинхронными методами, взаимодействующими между собой с использованием средств удаленного вызова методов (RMI - Remote Method Invocation), предоставляемых библиотекой System.Runtime.Remoting платформы .NET.

В случае языка MC#, программист может предусмотреть исполнение автономных асинхронных методов либо локально, либо удаленно. В последнем случае, метод может быть спланирован для исполнения на другой машине, выбираемой двумя способами: либо согласно явному указанию программиста (что не является типичным случаем), либо автоматически (обычно, на наименее загруженном узле кластера или машине Grid-сети). Взаимодействие асинхронных методов, в рамках языка MC#, реализуется посредством передачи сообщений с использованием каналов и обработчиков канальных сообщений. Эти каналы и обработчики определяются в MC#-программах с помощью связок.

Таким образом, написание параллельной, распределенной программы на языке MC# сводится к выделению с помощью специального ключевого слова async методов, которые должны быть исполнены асинхронно локально (в виде отдельных потоков), а также с помощью ключевого слова movable тех методов, которые могут быть перенесены для исполнения на другие машины. В лабораторной работе используются только асинхронные локальные методы.

**Модель программирования языка MC#: async-методы, каналы, обработчики- связки**

В любом традиционном языке объектно-ориентированного программирования, таком, как например, C#, обычные методы являются синхронными - вызывающая программа всегда ожидает завершения вычислений вызванного метода, и только затем продолжает свою работу.

При исполнении программы на параллельной архитектуре, сокращение времени её работы может быть достигнуто путем распределения множества исполняемых методов на несколько ядер одного процессора, и, возможно, отправкой части из них на другие процессоры (машины) при распределенных вычислениях.

Разделение всех методов в программе на обычные (синхронные) и асинхронные производится программистом с использованием специального ключевого слова async. Это единственное средство создания параллельных процессов (потоков) в языке MC#.

Кроме средств создания параллельных процессов, любой язык параллельного программирования должен содержать конструкции

* для обеспечения взаимодействия параллельных процессов между собой,
* для их синхронизации.

Основой взаимодействия параллельных процессов в языке MC# является передача сообщений. В языке MC#, средства взаимодействия между процессами оформлены в виде специальных синтаксических категорий - каналов и обработчиков канальных сообщений. При этом, синтаксически посылка сообщения по каналу или прием из него с помощью обработчика выглядят в языке как вызовы обычных методов.

Для синхронизации параллельных процессов в MC# используются связки (chords).

**Async-методы**

Общий синтаксис определения async-методов в языке MC# следующий:

модификаторы async имя\_метода ( аргументы )

{

< тело метода >

}

Ключевое слово async располагаеются на месте типа возвращаемого значения, поэтому синтаксическое правило его задания при объявлении метода в языке MC# имеет вид:

return-type ::= type | void | async

Задание ключевого слова async означает, что при вызове данного метода он будет запущен в виде отдельного потока локально, т.е., на данной машине (возможно, на отдельном ядре процессора), но без перемещения на другую машину.

Отличия async--методов от обычных методов состоят в следующем:

* вызов async-методов заканчивается, по существу, мгновенно,
* эти методы никогда не возвращают результаты.

Соответственно, согласно правилам корректного определения async-методов:

* они не могут объявляться статическими,
* в их теле не может использоваться оператор return.

**Каналы и обработчики канальных сообщений**

Каналы и обработчики канальных сообщений являются средствами для организации взаимодействия параллельных распределенных процессов между собой. Синтаксически, каналы и обработчики обычно объявляются в программе с помощью специальных конструкций - связок (chords).

В общем случае, синтаксические правила определения связок в языке MC# имеют вид:

chord-declaration ::= [handler-header] [ & channel-header ]\* body

handler-header ::= attributes modifiers handler handler-name

return-type ( formal-parameters )

channel-header ::= attributes modifiers channel channel-name

( formal-parameters )

Связки определяются в виде членов класса. По правилам корректного определения, каналы и обработчики не могут иметь модификатора static, а потому они всегда привязаны к некоторому объекту класса, в рамках которого они объявлены.

Обработчик используется для приема значений (возможно, предобработанного с помощью кода, являющегося телом связки) из канала (или группы каналов), совместно определенных с этим обработчиком. Если к моменту вызова обработчика связанный с ним канал пуст (т.е., по этому каналу значений не поступало или они все были выбраны посредством предыдущих обращений к обработчику), то этот вызов блокируется. Когда по каналу приходит очередное значение, то происходит исполнение тела связки (которое может состоять из произвольных вычислений) и по оператору return происходит возврат результирующего значения обработчику.

Наоборот, если к моменту прихода значения по каналу нет вызовов обработчика, то это значение просто сохраняется во внутренней очереди канала, где, в общем случае, накапливаются все сообщения, посылаемые по данному каналу. При вызове обработчика и при наличии значений во всех каналах соответствующей связки, для обработки в теле этой связки будут выбраны первые по порядку значения из очередей каналов.

Следует отметить, что, принципиально, срабатывание связки, состоящей из обработчика и одного или нескольких каналов, возможно в силу того, что они вызываются, в типичном случае, из различных потоков.

Вторая ключевая особенность языка MC# состоит в том, что каналы и обработчики могут передаваться в качестве аргументов методам (в том числе, async-методам) отдельно от объектов, которым они принадлежат.

Например, объявление канала sendInt для передачи одиночных целочисленных значений вместе с соответствующим обработчиком getInt для получения значений из этого канала, выглядит следующим образом:

****

Таким образом, мы можем послать целое число n по каналу sendInt, записав выражение

a.sendInt( n );

где a есть объект для которого определен канал sendInt.

Если имя канала использовано без уточняющих префиксов, например, как

c( n );

то подразумевается, как обычно, использование текущего объекта:

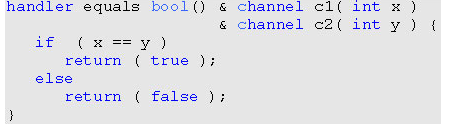
this.c ( n );

Обработчик используется для приема значений (возможно, предобработанного с помощью кода, являющегося телом связки) из канала (или группы каналов), совместно определенных с этим обработчиком. Например, для приема значения из канала sendInt можно записать

int m = a.getInt();

**Синхронизация в языке MC#**

В одной связке можно определить несколько каналов. Такого вида связки являются главным средством синхронизации параллельных потоков в языке MC#:

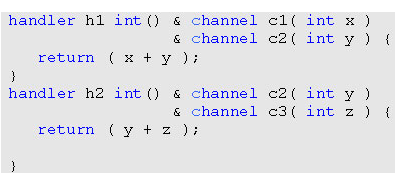


Таким образом, общее правило срабатывания связки состоит в следующем: тело связки исполняется только после того, как вызваны все методы из заголовка этой связки.

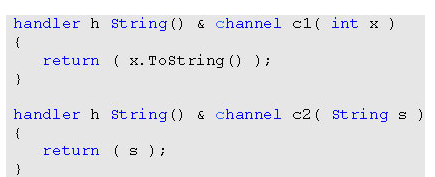
Приведенный выше пример иллюстрирует случай, когда один обработчик объявлен для нескольких каналов.

Также, возможно объявление канала, разделяемого между несколькими обработчиками:

Синтаксически, такое объявление возможно путем применения двух связок, как показано в примере ниже:



Наконец, возможен случай, когда один и тот же обработчик объявлен в разных связках и с различными каналами:



В этом случае, вызов обработчика при наличии значения хотя бы в одном из каналов, приведет к срабатыванию соответствующей связки. Легко заметить, что и здесь наличие значения более, чем в одном канале, может стать источником недетерминизма в поведении программы.

При использовании связок в языке MC# нужно руководствоваться следующими правилами их корректного определения:

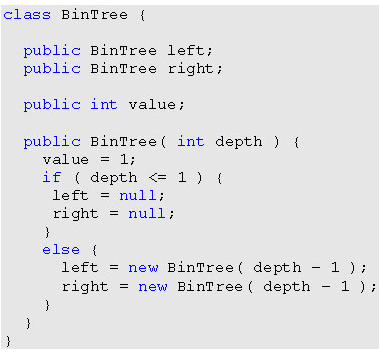
1. Формальные параметры каналов и обработчиков не могут содержать модификаторов ref или out.
2. Если в связке объявлен обработчик с типом возвращаемого значения return-type, то в теле связки должны использоваться операторы return только с выражениями, имеющими тип return-type.
3. Все формальные параметры каналов и обработчика в связке должны иметь различные идентификаторы.
4. Каналы и обработчики в связке не могут быть объявлены как static

**Примеры программирования на языке MC#**

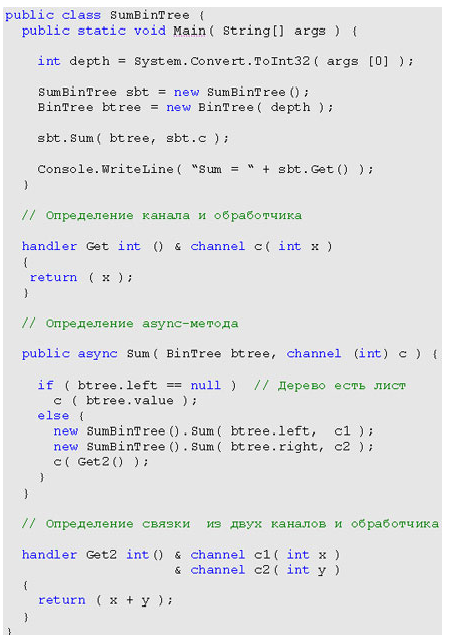
**Обход двоичного дерева**

Если структура данных задачи организована в виде дерева, то его обработку легко распараллелить путем обработки каждого поддерева отдельном async- методом.

Предположим, что мы имеем следующее определение бинарного дерева в виде класса BinTree:



Тогда просуммировать значения, находящиеся в узлах такого дерева можно с помощью следующей программы:



**Вычисление частичных сумм массива**

Рассмотрим более сложный пример использования обработчиков для организации конвейера между процессами, представленными async -методами.

Рассмотрим задачу вычисления частичных сумм массива f длины n.

А именно, по заданному массиву чисел f [ 0 : n-1 ] необходимо построить массив h [ 0 : n-1 ], такой что

Идея параллельного решения этой задачи состоит в разбиении массива f на p сегментов, где n кратно p, с дальнейшей одновременной обработкой этих сегментов данных длины m = n div p. Таким образом, обработка каждого сегмента будет производиться async -методом.

Разбиение исходного массива f на p сегментов производится таким образом, что в сегмент q, где ( 0 <= q < p ) попадают элементы f [ i ], такие что i mod p = q.

Так, например, если n = 16 и p = 4, то

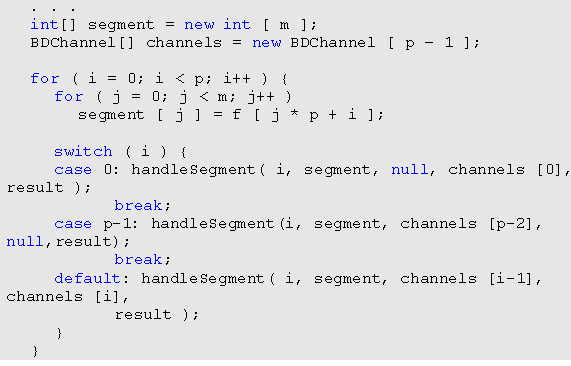
0 -ой сегмент составят числа f [ 0 ], f [ 4 ], f [ 8 ], f [ 12 ] ;

1 -ый сегмент составят числа f [ 1 ], f [ 5 ], f [ 9 ], f [ 13 ]

и т.д.

Параллельный алгоритм вычисления частичных сумм будет устроен так, что q -му процессу ( async -методу), обрабатывающему q -ый сегмент данных, достаточно будет общаться лишь с его соседями слева и справа (соответственно, 0 -му процессу - лишь с соседом справа, а последнему, (p-1) -му процессу - лишь с соседом слева) и главной программой для возврата результатов. Процесс с номером q будет вычислять все элементы h [j] результирующего массива, такие что j mod p = q, где 0 <= j < n.

Фрагмент главной программы, разбивающей исходный массив на сегменты и вызывающий async -метод handleSegment, показан ниже. Здесь первым аргументом этого метода является номер сегмента, а последним - имя канала для возврата результатов.



Класс BDChannel описывается следующим образом :

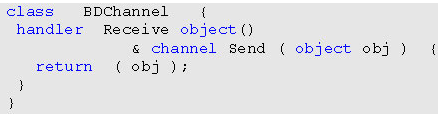
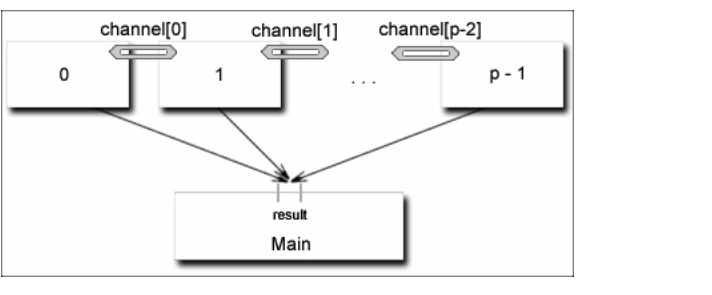


Схема взаимодействия процессов (async-методов) между собой и главной программой показана ниже:

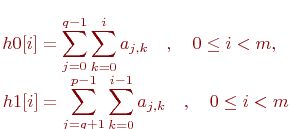


После разбиения, исходный массив f приобретает вид двумерной матрицы, распределенной по p процессам:



Другими словами, эта матрица получена из массива f разрезанием его на p сегментов и транспонированием каждого сегмента.

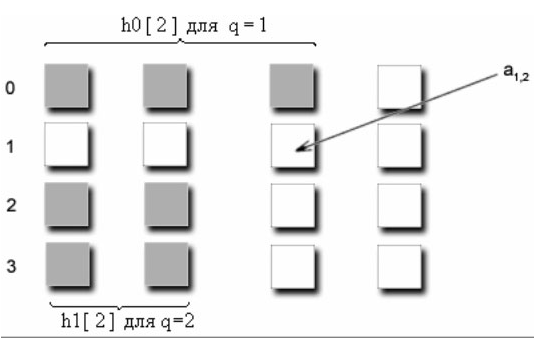
Ключевая идея алгоритма отдельного процесса q состоит в заполнении локальных для него массивов h0 и h1 (оба, имеющие размерность m ) в соответствии с формулами:



Это означает, что для процесса с номером q i-ый элемент массива h0 есть сумма всех элементов приведенной выше матрицы, которые расположены выше и слева элемента aq,i (включая и элементы столбца i ).

Аналогично, i-ый элемент массива h1 есть сумма всех элементов матрицы, которые расположены ниже и слева элемента aq,i (но, не включая элементов из столбца i ).

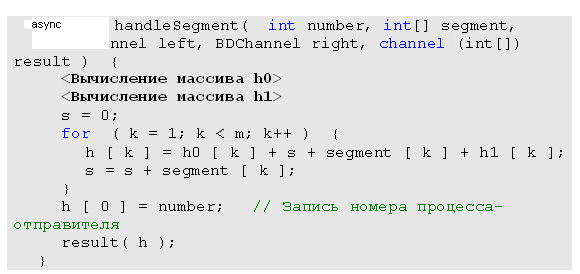
Ниже показана иллюстрация этого принципа для n = 16, p = 4 и q = 1, i = 2.



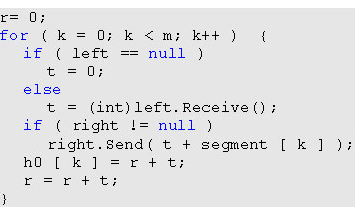
После того, как вычислены массивы h0 и h1 (посредством взаимодействия с соседними процессами), процесс с номером q может вычислить элемент h[i \* p + q] результирующего массива как



Получаемые результирующие m значений процесс q сохраняет в локальном массиве h для передачи их главной программе. Тогда общая схема async-метода handleSegment выглядит следующим образом:



Фрагмент программы, вычисляющий массив h0, приведен ниже.



**Вычисление чисел Фибоначчи**

Последовательность чисел Фибоначчи есть бесконечный ряд из натуральных чисел

a0, a1, a2, a3, . . .

таких, что

a0 = 1,

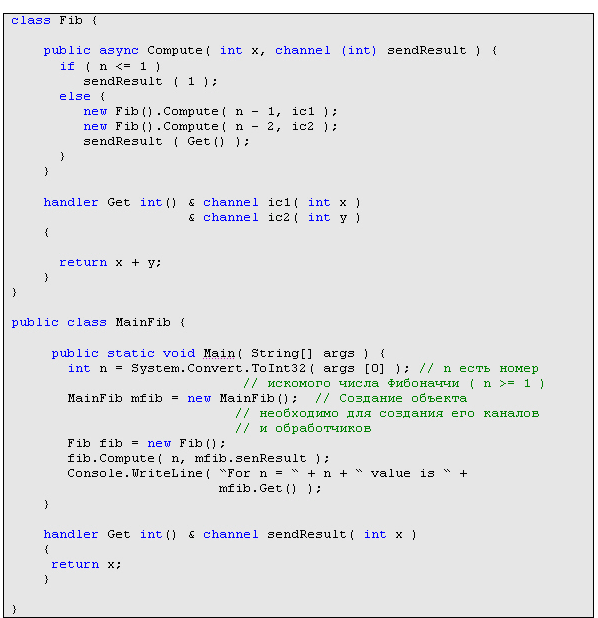
a1 = 1, и

ai = ai-1 + ai-2, для i >= 2.

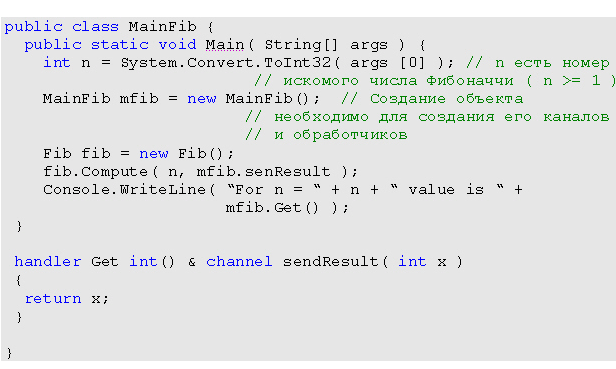
Построим параллельную программу, находящую n -ое ( n >= 0 ) число в ряду Фибоначчи, т.е., элемент an последовательности.

Первый вариант такой программы будет иметь рекурсивную структуру, соответствующую формуле определения чисел Фибоначчи. Основной вычислительный метод этой программы будет объявлен асинхронным, и будет возвращать вычисленный результат по каналу, переданному ему в качестве одного из входных аргументов. С другой стороны, в рекурсивных вызовах внутри этого метода будут использоваться каналы для получения значений от методов, вызванных рекурсивно.

Класс Fib, содержащий основной вычислительный метод Compute, может иметь следующий вид:

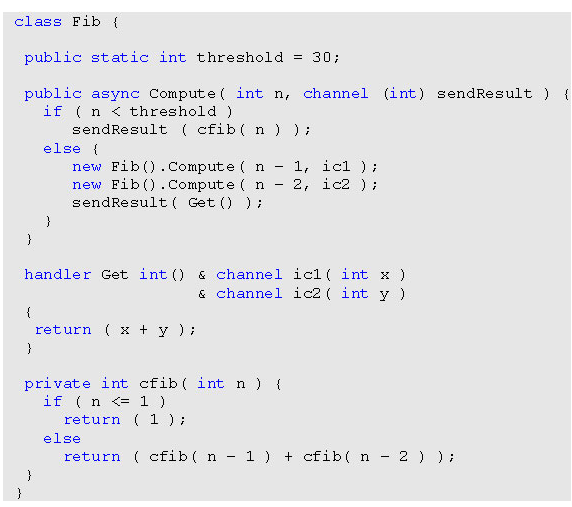


Главный класс для этой программы может выглядеть следующим образом:



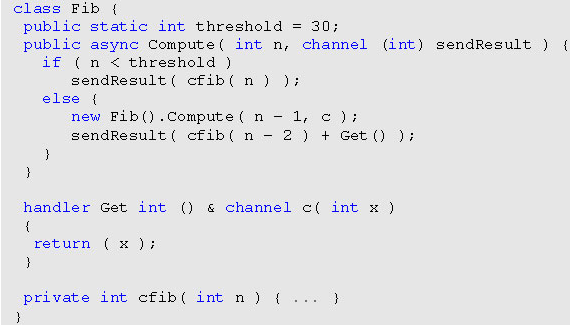
Легко видеть, что приведенный вариант параллельной программы является очень неэффективным, поскольку в нем порождается 2n асинхронных вызовов метода Compute, каждый из которых выполняет очень мало вычислительных операций: фактически, порождает два дополнительных вызова и передает результат по каналу. Очевидно, что в этом случае эффект от параллельного исполнения методов будет перекрыт накладными расходами на их порождение.

Избежать порождения асинхронных вызовов функции Compute для малых по величине аргументов n можно путем введения "локального" вычисления соответствующей функции для малых n:



Приведенный выше вариант является более эффективным, чем первый, но и он обладает существенным недостатком: теперь async-методы для больших n выполняют очень мало вычислительных операций.

Сократить общее количество порождаемых рекурсивно async-методов и более равномерно распределить по ним вычислительную нагрузку позволяет следующий, "линейный" вариант программы Fib:



**Построение списка простых чисел методом "решета Эратосфена"**

Рассмотрим параллельную программу построения списка простых чисел методом просеивания (другое название этого метода - "решето Эратосфена").

По условию задачи, по заданному натуральному числу N >= 2, необходимо найти все простые числа в интервале от 2 до N.

Метод просеивания состоит из следующих шагов:

1. из исходного списка l0 всех натуральных чисел от 2 до N

l0 = [2, … , N] выбирается первое число этого списка, а именно 2, и выдается в качестве первого выходного значения;

1. затем строится новый список l1, который получается из списка l0 вычеркиванием из него всех чисел, кратных очередному выбранному простому числу - на первом шаге, числу 2:

l1 = [3, 5, 7, … , N] (в предположении, что N нечетно)

1. затем данная процедура повторяется рекурсивно для вновь построенного списка.

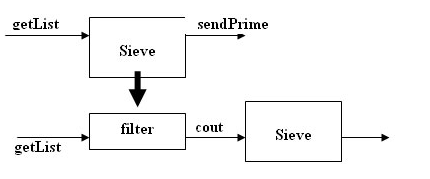
Алгоритм заканчивает свою работу, когда очередной список оказывается пустым.

В соответствии со сказанным выше, основная вычислительная процедура программы (метод Sieve), тем самым, должна производить следующие действия:

1. переслать первый элемент из входного потока натуральных чисел (если он не пуст) в выходной поток;
2. отфильтровать все оставшиеся числа их входного потока по модулю первого элемента и направить этот отфильтрованный поток на вход новой рекурсивно вызванной процедуре Sieve; выходным потоком для рекурсивно вызванной процедуры становится исходный выходной поток.

Выходным же каналом для этой новой процедуры будет исходный выходной канал.

Графически, один шаг рекурсивного развертывания программы может быть представлен следующим образом :

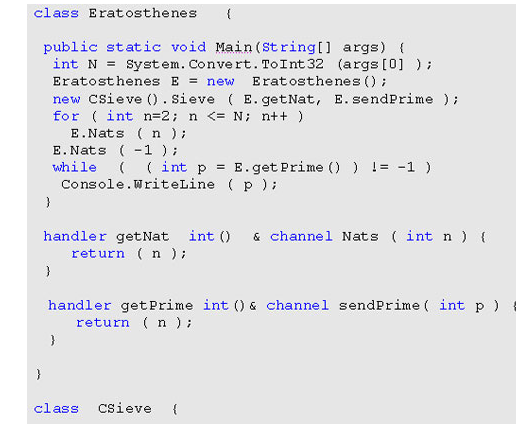
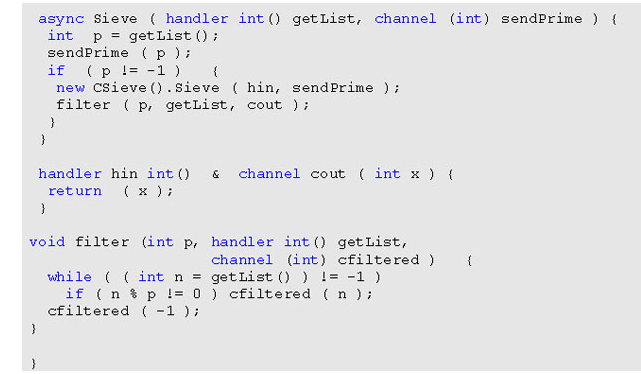


Рекурсивные вызовы метода Sieve сопровождаются созданием соответствующих объектов (класса CSieve ), имеющих каналы и обработчики. Эти каналы и обработчики используются для создания цепочки обрабатывающих элементов, с помощью которых производится просеивание потока натуральных чисел.

Программа состоит из следующих методов:

* async Sieve( handler int() getList, channel ( int ) sendPrime) - async-метод класса CSieve, который реализует собственно алгоритм просеивания: из обработчика getList предыдущего объекта цепочки читается поток чисел, фильтруется по модулю первого элемента из потока, и выбранные простые числа посылаются в канал sendPrime (через дальнейшие рекурсивные вызовы метода Sieve );
* void filter ( int p, handler int()getList, channel ( int ) cfiltered) - функция класса CSieve, реализующая фильтрацию потока натуральных чисел: из потока чисел, получаемых путем вызова обработчика getList, удаляются все элементы, которые делятся на число p, и посылаются по каналу cfiltered;
* public static void Main( String [] args )- главная функция из класса Eratosthenes, которая создает поток натуральных чисел Nats, и, по мере получения простых чисел из обработчика getPrime, выводит их на консоль.

Полный текст программы приведен ниже

Трансляция MC# - программ осуществляется командой:

mcsc <имя программы>.mcs

Запуск на выполнение осуществляется

<имя программы> <входные параметры>