PLINQ

PLINQ (Паралельний LINQ) - це розпаралелена реалізація LINQ. Запити PLINQ схожі з не паралельними запитами LINQ, основна відмінність цих запитів полягає в тому, що PLINQ повністю використовує можливості всіх процесорів в системі. Результати запитів PLINQ, також як і послідовні запити LINQ, повертають тип IEnumerable <T>, даний результат досягається шляхом поділу джерела даних на сегменти і паралельної обробки запиту кожного сегмента в окремому робочому потоці на декількох процесорах.

Завдяки PLINQ в значній мірі збільшується продуктивність в порівнянні із застарілим кодом. Однак паралелізм може привести до появи власних складнощів, і не всі операції запитів виконуються швидше в PLINQ.

Всі основні методи PLINQ міститися в класі System.Linq.ParallelEnumerable - простору імен System.Linq компілюються в збірку System.Core.dll. Цей клас включає в себе, реалізації всіх стандартних операторів запитів, підтримуваних LINQ to Objects.

Крім стандартних операторів-запитів клас ParallelEnumerable містить набір методів, які забезпечують поведінки, характерні для паралельного виконання. Ці методи представлені в Табл. 1.

Методи класу ParallelEnumerable	Опис
AsParallel	Точка входу для PLINQ. Вказує на необхідність паралелізації решти запиту, якщо це можливо.
AsSequential<TSource>	Вказує на необхідність послідовного виконання решти запиту як непаралельного запиту LINQ.
AsOrdered	Вказує, що PLINQ повинен зберегти порядок вихідної послідовності для іншої частини запиту або до тих пір, поки порядок не буде змінений.
AsUnordered<TSource>	Вказує, що PLINQ не повинен зберігати порядок вихідної послідовності для іншої частини запиту.
WithCancellation<TSource>	Вказує, що PLINQ повинен періодично відстежувати стан наданого токена скасування і скасовувати виконання при запиті.
WithDegreeOfParallelism<TSource>	Вказує максимальну кількість процесорів, яке повинен використовувати PLINQ для паралелізації запиту.
WithMergeOptions<TSource>	Надає підказку про те, як PLINQ повинен, якщо це можливо, виконувати злиття паралельних

	результатів в одну послідовність в потоці-споживачі.
WithExecutionMode<TSource>	Вказує, чи повинен PLINQ виконувати паралелізацію запиту, навіть якщо згідно поведінки за замовчуванням він буде виконуватися послідовно.
ForAll<TSource>	Багатопотоковий метод перерахування, який на відміну від ітерації результатів запиту дозволяє обробляти результати паралельно без попереднього злиття в потік-споживач.
Перевантаження Aggregate	Унікальне для PLINQ перевантаження, що забезпечує проміжне агрегування локальних частин потоку, і надає функцію остаточного агрегування для об'єднання результатів всіх частин.

Метод AsParallel

Метод AsParallel () - це деяка вхідна точка у використанні запитів PLINQ. Він перетворює послідовність даних в ParallelQuery. Механізм LINQ виявляє використання ParallelQuery як джерело в запиті і переключається на виконання PLINQ автоматично. Метод має AsParallel() кілька перевантажених варіантів, які представлені нижче:

Перший працює з використанням IEnumerable <TSource> і повертає екземпляр ParallelQuery <TSource>, який може бути використаний в якості основи запиту PLINQ:

```
public static ParallelQuery <TSource> AsParallel <TSource> (
this IEnumerable <TSource> source
)
```

Другий тип створює екземпляр ParallelQuery з IEnumerable і призначений для підтримки успадкованих колекцій, таких як System.Collections.ArrayList. Об'єкт ParallelQuery не є строго типізований і не може використовуватися як основа запиту PLINQ без перетворення в ParallelQuery:

```
public static ParallelQuery AsParallel (
this IEnumerable source
)
```

Обидва типи повертають виключення виду ArgumentNullException.

Приклад використання PLINQ запиту

Щоб перетворити LINQ в PLINQ, в запиті LINQ досить викликати метод AsParallel () в LINQ запиті. У наступному прикладі знаходяться числа в діапазоні від 1 до 100000, кратні двом за допомогою неоптимізованого паралельного алгоритму:

```
var source = Enumerable.Range(1, 100000);
            var parallelQuery = from num in source.AsParallel()
                                where num % 2 == 0
                                select num;
  foreach (var n in parallelQuery)
            {
                Console.WriteLine(n);
            }
            Console.ReadLine();
```

Цей же запит можна було записати з використанням лямбда-вирази:

```
var parallelQuery = source.AsParallel().Where(n => n % 2 == 0).Select(n => n);
```

Класи паралельних колекцій

У версії .NET Framework 4.0 став доступний новий простір імен System.Collections.Concurrent, який містить кілька класів колекцій, які є потокобезпечними і масштабованими. Це означає, що кілька потоків можуть безпечно і ефективно додавати і видаляти елементи з таких колекцій, не вимагаючи при цьому додаткової синхронізації в призначеному для користувача коді. Паралельні колекції відрізняються від стандартних колекцій і тим, що вони містять спеціальні методи для виконання атомарних операцій типу "перевірити-і-виконати" (методи TryPop, TryAdd). У Табл. 2 перераховані нові класи паралельних колекцій, які були додані в .NET Framework 4.0.

Таблиця 2. Короткий опис класів паралельних колекцій

Класс	Описание
`BlockingCollection<T>`	Надає можливості блокування і обмеження для потокобезпечних колекцій, що реалізують IProducerConsumerCollection <T>. Потоки-виробники блокуються, якщо слоти відсутні або колекція є повною. Потоки-споживачі блокуються, якщо колекція порожня. Цей тип також може не підтримувати блокуючий доступ споживачів і виробників. Колекцію BlockingCollection <T> можна використовувати в якості базового класу або резервного сховища для надання блокування і обмеження для будь-якого класу колекції, що підтримує IEnumerable <T>.
`ConcurrentBag<T>`	Потокобезпечна реалізація наборів, що надає масштабовані операції додавання і отримання.
`ConcurrentDictionary<TKey, TValue>`	Тип паралельного і масштабованого словника.
`ConcurrentQueue<T>`	Паралельна і масштабована черга FIFO.
`ConcurrentStack<T>`	Паралельний і масштабований стек LIFO.

Паралельні колекції зазвичай бувають, корисні при вирішенні тривіальних завдань багатопотоковості, коли потрібна потокобезпечна колекція. Слід пам'ятати ряд принципів при роботі з паралельними колекціями:

- Паралельні колекції слід використовувати в тих випадках, коли є сценарії з високою конкурентністю за ресурси комп'ютера. В іншому випадку використовуються звичайні колекції.

- Паралельні колекції не гарантують потокобезпечності;

- Якщо в процесі перебору елементів паралельної колекції інший потік її модифікує, виняток згенеровано не буде. Замість цього виходить колекція зі старим і новим вмістом;

- Не існує паралельної версії колекції List<T>;

- Паралельні класи стека, черги і набору (bag) всередині реалізовані на основі зв'язних списків. Це робить їх менш ефективними в плані споживання пам'яті в порівнянні з непаралельними версіями класів Stack і Queue, але кращими для паралельного доступу, оскільки зв'язні списки є відмінними кандидатами для lock-free або low-lock реалізацій.

Використання паралельних колекцій не еквівалентне використанню звичайних колекцій з операторами lock. Наприклад, використання паралельної колекції ConcurrentDictionary буде виконуватись повільніше в даному випадку:

```
var d = new ConcurrentDictionary <int, int> ();
for (int i = 0; i <1000000; i ++) d [i] = 123;
```

Ніж використання звичайної колекції Dictionary

```
var d = new Dictionary <int, int> ();
for (int i = 0; i <1000000; i ++) lock (d) d [i] = 123;
```

Якщо необхідно отримати значення з колекції ConcurrentDictionary, то операція виконується швидше, оскільки читання являюется lock-free.

Інтерфейс IProducerConsumerCollection <T>

Даний інтерфейс забезпечує уніфіковане представлення для колекцій виробників / споживачів, щоб абстракції вищого рівня, такі як BlockingCollection <T>, могли використовувати колекцію в якості базового механізму зберігання. Інтерфейс IProducerConsumerCollection <T> був доданий в версію .NET 4 для підтримки нових, безпечних щодо потоків класів колекцій. Існує два основні сценарії використання колекцій типу постачальник / споживач (producer / consumer):

- Додавання елементів (" постачання");

- Отримання елемента і його одночасне видалення ("споживання").

Синтаксис використовуваний при створенні інтерфейсу наведено нижче:

```
public interface IProducerConsumerCollection <T>: IEnumerable <T>, ICollection,
IEnumerable
```

де T - Визначає тип елементів колекції.

Класичним прикладом використання цього інтерфейсу є стеки і черги. Наступні класи реалізують цей інтерфейс:

```
ConcurrentStack <T>;
ConcurrentQueue <T>;
ConcurrentBag <T>.
```

Інтерфейс IProducerConsumerCollection <T> розширює інтерфейс ICollection <T> шляхом додавання методів і властивостей представлених в Табл. 3.

Ім’я	Опис
CopyTo(T[] array, int index);	Метод копіює елементи колекції IProducerConsumerCollection <T> в масив.
ToArray(T item)	Метод копіює елементи, що містяться в колекції IProducerConsumerCollection <T>, в новий масив.
TryAdd(T item)	Метод намагається додати об'єкт в колекцію IProducerConsumerCollection <T>.
TryTake(out T item)	Метод намагається видалити і повернути об'єкт з колекції IProducerConsumerCollection <T>.
IEnumerable.GetEnumerator()	Метод повертає нумератор, який виконує ітерацію за елементами колекції. (Успадкованих від IEnumerable.)
IEnumerator<T> GetEnumerator()	Метод повертає нумератор, що виконує перебір елементів в колекції. (Успадкованих від IEnumerable <T>.)
Count	Властивість повертає число елементів, що містяться в колекції ICollection. (Успадкованих від ICollection.)
IsSynchronized	Дана властивість отримує значення, що дозволяє визначити, чи є доступ до колекції ICollection синхронізованим (потокобезпечна).
SyncRoot	Властивість отримує об'єкт, який можна використовувати для синхронізації доступу до ICollection.

Методи TryAdd () і TryTake () перевіряють, чи може бути виконана операція додавання / видалення елемента, і якщо операція може бути виконана, то вона виконується.

Метод TryTake () повертає значення false, якщо колекція порожня. Метод TryAdd () завжди завершується успішно і повертає true у всіх трьох існуючих реалізаціях. Якщо ви напишете свою власну паралельну колекцію, яка буде забороняти дублікати, то вона зможе повертати false, якщо такий елемент вже існує в колекції.

Конкретний елемент, який видаляється при виклику методу TryTake (), визначається конкретною реалізацією:

У класі ConcurrentStack <T> - метод TryTake () видаляє останній доданий елемент;

У класі ConcurrentQueue <T> - метод TryTake () видаляє найперший доданий елемент;

У класі ConcurrentBag <T> - метод TryTake () видаляє будь-який елемент, який може бути видалений, максимально ефективно.

Ці класи реалізують методи TryTake () і TryAdd (), явно надаючи ту ж саму функціональність за допомогою інших відкритих методів з більш точними назвами, такими як TryDequeue () і TryPop ().

Приклад ConcurrentQueue

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using System.Collections;
namespace ConcurrentQueueExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // створюємо колекцію
            ConcurrentQueue<int> sharedQueue = new ConcurrentQueue<int>();
            // заповнюємо колекцію в циклі за допомогою методу Enqueue
            for (int i = 0; i < 1000; i++)
            {
                sharedQueue.Enqueue(i);
            }
            // оголошуємо змінну-лічильник кількості оброблених елементів
            int itemCount = 0;
            // створюємо список задач
            Task[] tasks = new Task[10];
            for (int i = 0; i < tasks.Length; i++)
            {
                // створюємо задачу
                tasks[i] = new Task(() =>
                {
                    while (sharedQueue.Count > 0)
                    {
                        Thread.Sleep(10);
                       // оголошуємо змінну для запитів видалення з черги
                        int queueElement;
                        // видаляємо елемент з колекції за допомогою методу TryDequeue
                        bool gotElement = sharedQueue.TryDequeue(out queueElement);
                        // збільшуємо значення змінної і зберігаємо результат
                         if (gotElement)
                        {
                        Interlocked.Increment(ref itemCount);
```

```
                    }
                }
                }
            });
            // запускаємо нову задачу
            tasks[i].Start();
        }
            // чекаємо завершення всіх завдань
            Task.WaitAll(tasks);
            // виводимо на екран звіт про кількість оброблених елементів
            Console.WriteLine("Оброблено елементів: {0}", itemCount);
             Console.ReadLine();
    }
}
}
```

Приклад ConcurrentStack

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using System.Collections;
namespace ConcurrentStackExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // створюємо колекцію
            ConcurrentStack<int> sharedStack = new ConcurrentStack<int>();
            // заповнюємо колекцію в циклі за допомогою методу Push
            for (int i = 0; i < 1000; i++)
            {
                sharedStack.Push(i);
            }
            // оголошуємо змінну-лічильник кількості оброблених елементів
            int itemCount = 0;
            // створюємо список задач
            Task[] tasks = new Task[10];
            for (int i = 0; i < tasks.Length; i++)
            {
                // створюємо задачу
                tasks[i] = new Task(() =>
                {
                    while (sharedStack.Count > 0)
                    {
                        Thread.Sleep(10);
                        int queueElement;
                        // видаляємо елемент з колекції за допомогою методу TryPop
                        bool gotElement = sharedStack.TryPop(out queueElement);
                        // збільшуємо значення змінної і зберігаємо результат
                        if (gotElement)
                        {
                        Interlocked.Increment(ref itemCount);
                        }
                    }
                });
                // запускаємо нову задачу
                tasks[i].Start();
            }
                // чекаємо завершення всіх задач
                Task.WaitAll(tasks);
                // виводимо на екран звіт про кількість оброблених елементів
                Console.WriteLine("Оброблено елементів: {0}", itemCount);
                 Console.ReadLine();
        }
    }
}
```

Приклад ConcurrentBag

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using System.Collections;
namespace ConcurrentCollection
{
    class Program
    {
        static void Main(string[] args)
        {
            // створюємо колекцію
            ConcurrentBag<int> sharedBag = new ConcurrentBag<int>();
            // заповнюємо колекцію в циклі за допомогою методу Add
            for (int i = 0; i < 1000; i++)
            {
                sharedBag.Add(i);
            }
            // оголошуємо змінну-лічильник кількості оброблених елементів
            int itemCount = 0;
            // створюємо список задач
            Task[] tasks = new Task[10];
            for (int i = 0; i < tasks.Length; i++)
            {
                // створюємо задачу
                tasks[i] = new Task(() =>
                {
                    while (sharedBag.Count > 0)
                    {
                        Thread.Sleep(10);
                        int queueElement;
                        // видаляємо елемент з колекції за допомогою методу TryTake
                        bool gotElement = sharedBag.TryTake(out queueElement);
                        // збільшуємо значення змінної і зберігаємо результат
                        if (gotElement)
                        {
                        Interlocked.Increment(ref itemCount);
                        }
                    }
                });
                // запускаємо нову задачу
                tasks[i].Start();
            }
                // чекаємо завершення всіх задач
                Task.WaitAll(tasks);
                // виводимо на екран звіт про кількість оброблених елементів
                Console.WriteLine("Оброблено елементів: {0}", itemCount);
                 Console.ReadLine();

        }
    }
}
```

ConcurrentBag можна уявити як набір черг з двостороннім доступом (deque). Кожен потік при роботі з колекцією звертається до своєї власної черги, додаючи і видаляючи елементи з її початку. Коли трапляється так, що черга одного потоку порожня, а йому потрібно витягти елемент, то він витягує його з черги сусіднього потоку, але вже не спочатку черги, а з протилежного кінця черги. Такий підхід дозволяє практично не перетинатися різним потокам за даними.

Приклад ConcurrentDictionary

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Collections.Concurrent;
using System.Threading.Tasks;
namespace ConcurrentDictionaryExample
{
    class BankAccount
    {
        public int Balance
        {
            get;
            set;
        }
        class Program
        {
            static void Main(string[] args)
            {
                // створюємо екземпляр банківського рахунку
                BankAccount account = new BankAccount();
                // створюємо колекцію
                ConcurrentDictionary<object, int> sharedDict
                    = new ConcurrentDictionary<object, int>();
                        // створюємо список завдань, які повертають цілочисельний масив
                Task<int>[] tasks = new Task<int>[10];
                for (int i = 0; i < tasks.Length; i++)
                {
                    // поміщаємо початкові значення в словник
                    sharedDict.TryAdd(i, account.Balance);
                    // создаем новую задачу
                    tasks[i] = new Task<int>((keyObj) =>
                    {
                        // створюємо змінну для використання в циклі
                        int currentValue;
                        bool gotValue;
                        // створюємо цикл для поновлення балансу рахунку
                        for (int j = 0; j < 1000; j++)
                        {
                            // отримуємо поточне значення зі словника
                            gotValue = sharedDict.TryGetValue(keyObj, out currentValue);
                            // збільшуємо значення і оновлюємо словник
                            sharedDict.TryUpdate(keyObj, currentValue + 1, currentValue);
                        }
                        // створюємо змінну кінцевого результату
                        int result;
                        // отримуємо результат зі словника
                        gotValue = sharedDict.TryGetValue(keyObj, out result);
                        // повертаємо значення результату, якщо є
                        if (gotValue)
                        {
                            return result;
                        }
                        else
                        {
                            // якщо немає результату - викликаємо виняток
                            throw new Exception(
                                String.Format("Немає елементів даних доступних для
об'єкта {0}", keyObj));
                        }
                    }, i);
                    // запускаємо задачу
                    tasks[i].Start();
                }
                // оновлюємо баланс рахунку за допомогою результатів виконання завдань
                for (int i = 0; i < tasks.Length; i++)
                {
                    account.Balance += tasks[i].Result;
                }
                // виводимо значення лічильника
                Console.WriteLine("Очікуване значення: {0}, Баланс: {1}",
                    10000, account.Balance);
                Console.ReadLine();
            }
        }
    }
}
```

Приклад BlockingCollection

Колекція BlockingCollection, являє собою потокобезпечну колекцію, яка здійснює блокування і очікує, поки не з'явиться можливість виконати дію по додаванню або вилученню елемента.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections.Concurrent;
using System.Threading.Tasks;
namespace BlockingCollectionExample
{
     class BankAccount {
        public int Balance {
            get;
            set;
        }
    }
    class Deposit {
        public int Amount {
            get;
            set;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
             // створюємо колекцію BlockingCollection
            BlockingCollection<Deposit> blockingCollection
                = new BlockingCollection<Deposit>();
            // створюємо і запускаємо завдання, яка буде генерувати депозити і поміщати
їх в колекцію
            Task[] producers = new Task[3];
            for (int i = 0; i < 3; i++) {
                producers[i] = Task.Factory.StartNew(() => {
                    // створюємо депозити
                    for (int j = 0; j < 20; j++) {
                     // створюємо перевід
                        Deposit deposit = new Deposit { Amount = 100 };
                     // поміщаємо перевід в колекцію
                        blockingCollection.Add(deposit);
                    }
                });
            };
            // створюємо продовження, яке буде сигналізувати про закінчення "поставки"
            Task.Factory.ContinueWhenAll(producers, antecedents => {
                // створюємо сигнал - "постачання" закінчено
                Console.WriteLine("Сигнал про закінчення виробництва");
                blockingCollection.CompleteAdding();
            });
          // створюємо банківський рахунок
            BankAccount account = new BankAccount();
          // створюємо споживача, який буде оновлювати баланс, заснований на депозитах
            Task consumer = Task.Factory.StartNew(() => {
                while (!blockingCollection.IsCompleted) {
                    Deposit deposit;
                    // намагаємося отримати наступний елемент колекції
                    if (blockingCollection.TryTake(out deposit)) {
                        // оновлюємо баланс з урахуванням суми переказу
                        account.Balance += deposit.Amount;
                    }
                }
                // виводимо фінальний баланс
                Console.WriteLine("Підсумковий баланс: {0}", account.Balance);
            });
            consumer.Wait();
            Console.ReadLine();
        }
    }
}
```

Сигнальні повідомлення

Сигнальні повідомлення дозволяють реалізувати різні схеми синхронізації, як взаємне виключення, так і умовну синхронізацію. При умовній синхронізації потік блокується в очікуванні події, яка генерується в іншому потоці. Платформа .NET надає три типи сигнальних повідомлень: AutoResetEvent, ManualResetEvent і ManualResetEventSlim, а також шаблони синхронізації, побудовані на сигнальних повідомленнях (CountdownEvent, Barrier). Перші два типи побудовані на об'єкті ядра операційної системи. Третій тип ManualResetEventSlim є полегшеною версією об'єкта ManualResetEvent, є більш продуктивними.

У наступному фрагменті два потоки використовують один і той же об'єкт типу ManualResetEvent. Перший потік виводить повідомлення від другого потоку. Повідомлення записується в розподілену змінну. Виклик методу WaitOne блокує перший потік в очікуванні сигналу від другого потоку. Сигнал генерується при виклику методу Set.

```
void OneThread(object o)
{
  AutoResetEvent are = (AutoResetEvent)o;
  are.WaitOne();
  Console.WriteLine("Data from thread #2: " + data);
}
void SecondThread(object o)
{
  AutoResetEvent are = (AutoResetEvent)o;
  Console.WriteLine("Writing data");
  data = "BBBBBB";
  are.Set();
}
```

Завдання

1. Створити колекцію класів машина та за допомогою запитів PLINQ вивести тільки значення поля номерний знак для всіх машин колекції. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentQueue, використовуючи примітив синхронізації AutoResetEvent.

2. Створити колекцію класів машина та за допомогою запитів PLINQ вивести тільки машини 2000-го року випуску і вище. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentStack, використовуючи примітив синхронізації AutoResetEvent**.**

3. Створити колекцію класів машина та за допомогою запитів PLINQ вивести тільки машини марки “BMW”. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentBag, використовуючи примітив синхронізації AutoResetEvent**.**

4. Створити колекцію класів працівник та за допомогою запитів PLINQ вивести всіх працівників у зворотному порядку. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentDictionary, використовуючи примітив синхронізації AutoResetEvent**.**

5. Створити колекцію класів працівник та за допомогою запитів PLINQ вивести всіх працівників чий рік народження кратний двійці. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу BlockingCollection, використовуючи примітив синхронізації AutoResetEvent**.**

6. Створити колекцію класів працівник та за допомогою запитів PLINQ вивести імена всіх працівників у верхньому регістрі. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentQueue, використовуючи примітив синхронізації ManualResetEvent.

7. Створити колекцію класів працівник та за допомогою запитів PLINQ вивести імена та посади всіх працівників, які старші 20 і молодші 30 років. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentStack, використовуючи примітив синхронізації ManualResetEvent**.**

8. Створити колекцію класів комп'ютер та за допомогою запитів PLINQ вивести інформацію про всі коп'ютери (всі поля). Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentBag, використовуючи примітив синхронізації ManualResetEvent**.**

9. Створити колекцію класів комп'ютер та за допомогою запитів PLINQ вивести моделі коп'ютерів за абеткою у порядку зростання. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу ConcurrentDictionary, використовуючи примітив синхронізації ManualResetEvent**.**

10. Створити колекцію класів комп'ютер та за допомогою запитів PLINQ вивести моделі коп'ютерів за абеткою у порядку спадання. Змоделювати задачу Виробник / Споживач із одним споживачем і одним виробником, що розділяють паралельну колекцію типу BlockingCollection, використовуючи примітив синхронізації ManualResetEvent**.**