1. Что такое *TPL*? Как и для чего используется тип *Task*

Библиотека параллельных задач TPL (Task Parallel Library) **System.Threading.Tasks**. Данная библиотека упрощает работу с многопроцессорными, многоядерными системами и по созданию новых потоков.

Task – абстракция более высокого уровня, чем поток.

Task класс описывает отдельную продолжительную операцию, которая запускается асинхронно в одном из потоков из пула потоков (можно запускать синхронно в текущем потоке).

2. Почему эффект от распараллеливания наблюдается на большом количестве элементов?

**При малом объеме элементов массива, накладные расходы, связанные с организацией многопоточной обработки, превышают выигрыш от параллельности обработки.**

3. В чем основные достоинства работы с задачами по сравнению с потоками?

Вызов Start() для Task не создает новый поток, а помещает задачу в очередь готовых задач – пул потоков. Планировщик (TaskScheduler) в соответствии со своими правилами распределяет готовые задачи по рабочим потокам. Действия планировщика можно корректировать с помощью параметров задач. Момент фактического запуска задачи в общем случае не определен и зависит от загруженности пула потоков.

►Поток (Thread) – это низкоуровневый инструмент для организации параллельной работы

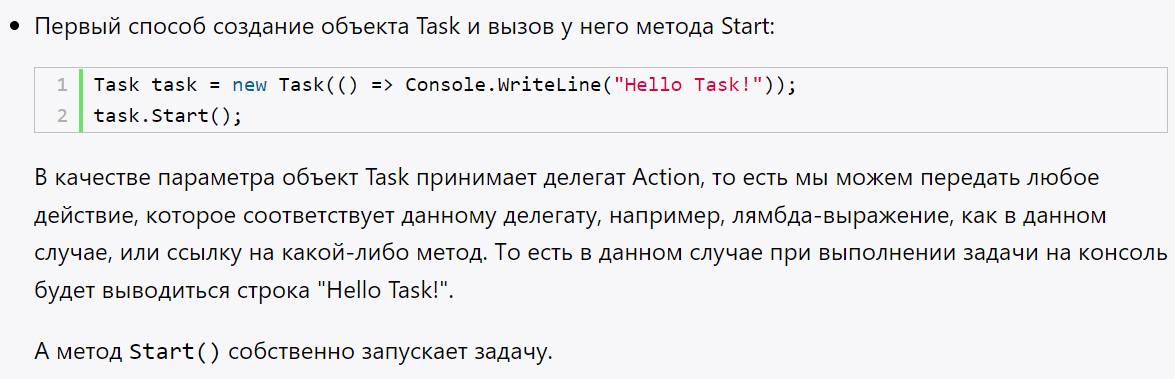
►Ограничения:

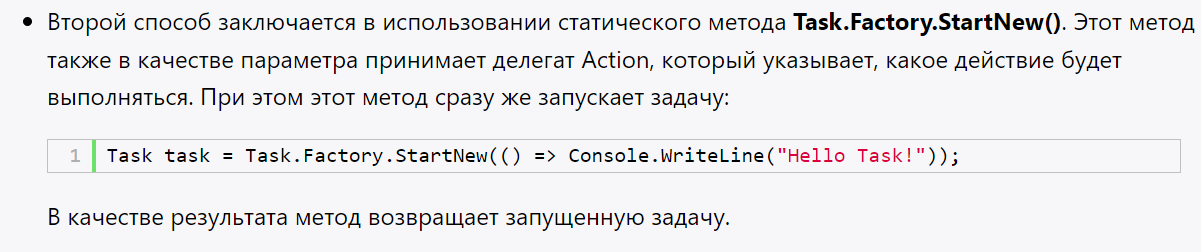
♣ 1) отсутствует механизм продолжений (после завершения метода, работающего в потоке, в этом же потоке автоматически запускается другой заданный метод)

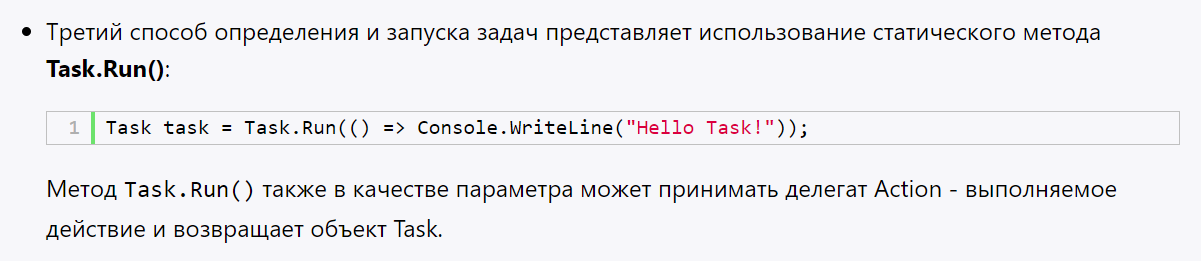
♣ 2) Затруднено получение значения результата из потока

♣ 3) создание множества потоков ведёт к повышенному расходу памяти и замедлению работы приложения

4. Приведите три способа создания и/или запуска Task?



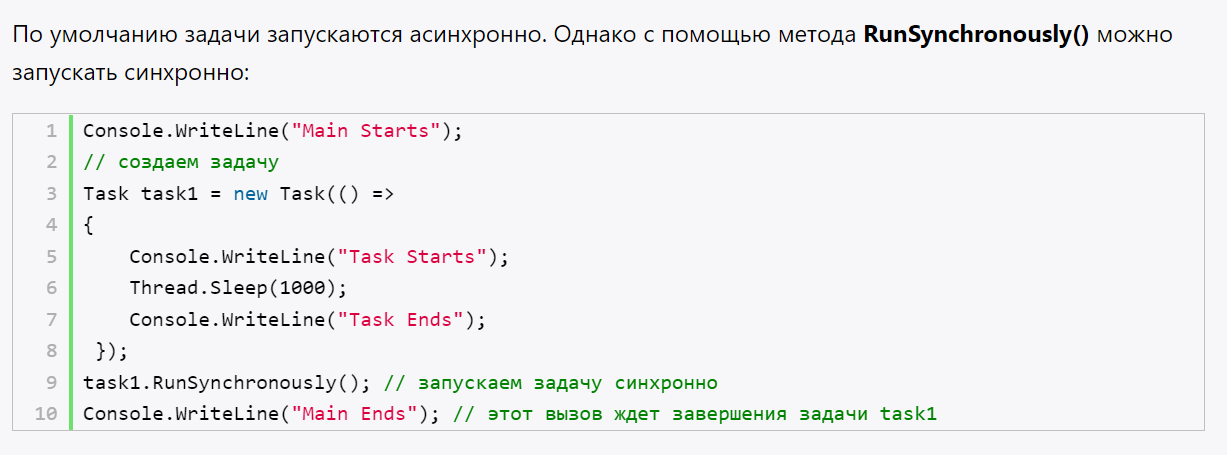




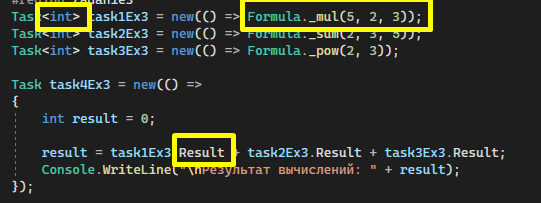
5. Как и для чего используют методы *Wait*(), *WaitAll*() и *WaitAny*()?

**Wait()** – приостанавливает текущий поток до завершения задачи  
**WaitAll()** – приостанавливает текущий поток до завершения всех указ. задач  
**WaitAny()** – приостанавливает текущий поток до завершения любой из указ. задач

6. Приведите пример синхронного запуска *Task*?

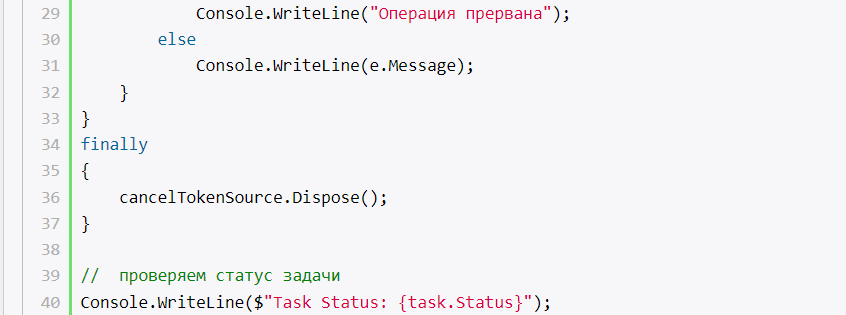


7. Как создать задачу с возвратом результата?



\*Указываем возвращаемый тип данных, затем получаем результат задачи при помощи свойства Result.

8. Как обработать исключение, если оно произошло при выполнении *Task*?

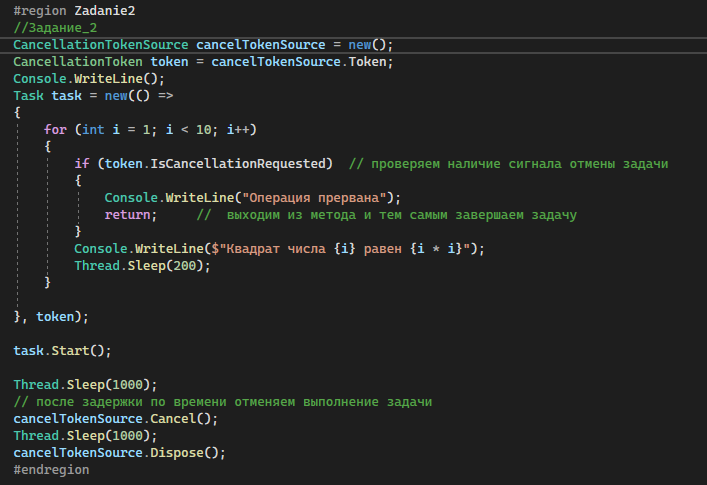


9. Что такое *CancellationToken* и как с его помощью отменить выполнение задач?

**CancellationToken – структура, предназначенная для прерывания выполняемой задачи.** System.Threading.

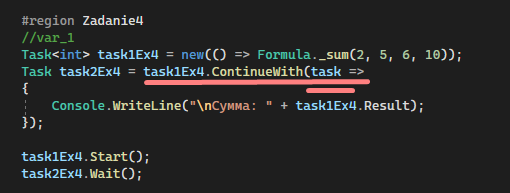
Общий алгоритм отмены задачи обычно предусматривает следующий порядок действий:

1. Создание объекта **CancellationTokenSource**, который управляет и посылает уведомление об отмене токену.
2. С помощью свойства **CancellationTokenSource.Token** получаем собственно токен - объект структуры **CancellationToken** и передаем его в задачу, которая может быть отменена.
3. Определяем в задаче действия на случай ее отмены.
4. Вызываем метод **CancellationTokenSource.Cancel()**, который устанавливает для свойства **CancellationToken.IsCancellationRequested** значение true. Стоит понимать, что сам по себе метод **CancellationTokenSource.Cancel()** не отменяет задачу, он лишь посылает уведомление об отмене через установку свойства CancellationToken.IsCancellationRequested. Каким образом будет происходить выход из задачи, это решает сам разработчик.
5. Класс CancellationTokenSource реализует интерфейс IDisposable. И когда работа с объектом CancellationTokenSource завершена, у него следует вызвать метод Dispose для освобождения всех связанных с ним используемых ресурсов. (Вместо явного вызова метода Dispose можно использовать конструкцию using).



10. Как организовать задачу продолжения (continuation task) ?

Позволяют определить задачи, которые выполняются после завершения других задач. Благодаря этому мы можем вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.



11. Как и для чего используется объект ожидания при создании задач продолжения?

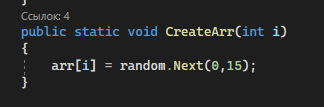
Метод Wait() приостанавливает поток для ожидания завершения задачи. Вызов метода для простой задачи вызывает исключение.

12. Поясните назначение класса *System.Threading.Tasks.Parallel*?

Класс **Parallel** также является частью TPL и предназначен для упрощения параллельного выполнения кода. **Parallel** имеет ряд методов, которые позволяют распараллелить задачу.

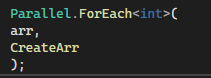
13. Приведите пример задачи с *Parallel.For(int, int, Action<int>)*

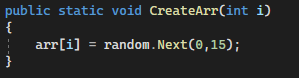




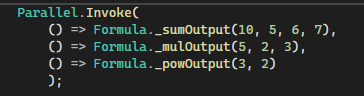
14. Приведите пример задачи с *Parallel.ForEach*





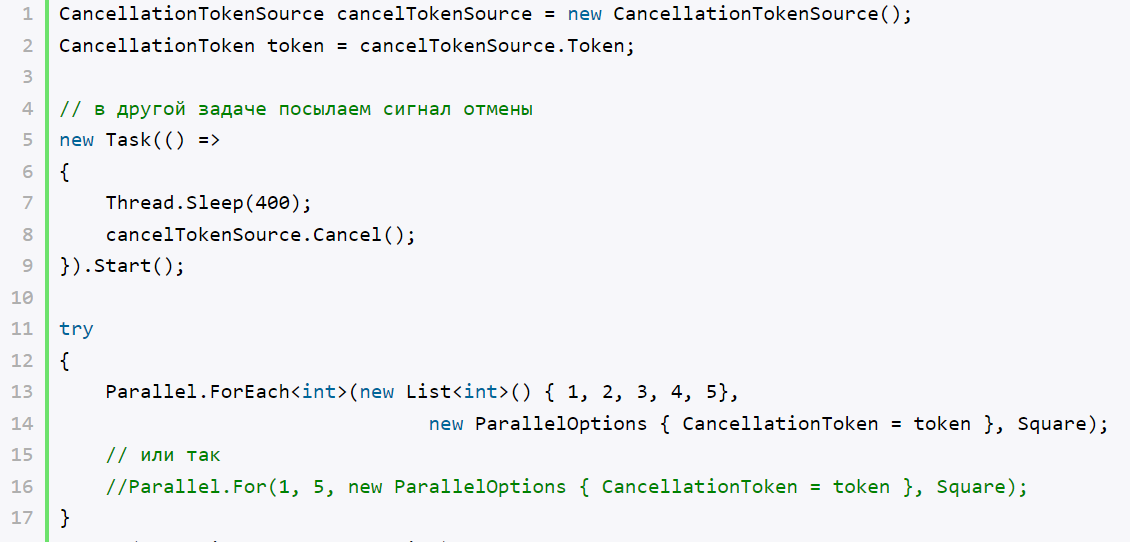


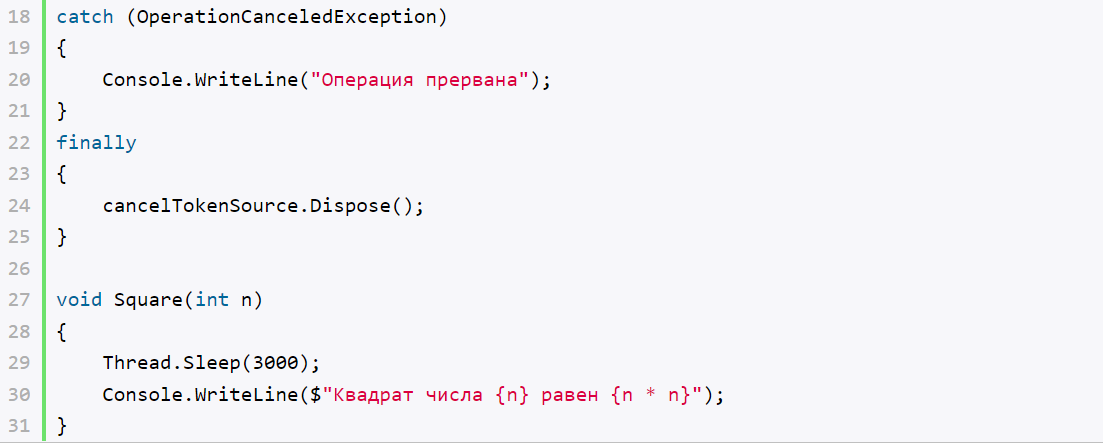
15. Приведите пример с *Parallel.Invoke()*



\*Вызов нескольких методов

16. Как с использованием *CancellationToken* отменить параллельные операции?

Для отмены выполнения параллельных операций, запущенных с помощью методов **Parallel.For()** и **Parallel.ForEach()**, можно использовать перегруженные версии данных методов, которые принимают в качестве параметра объект **ParallelOptions**. Данный объект позволяет установить токен: 



В параллельной запущенной задаче через 400 миллисекунд происходит вызов cancelTokenSource.Cancel(), в результате программа выбрасывает исключение OperationCanceledException, и выполнение параллельных операций прекращается.

17. Для чего используют *BlockingCollection*<T>, в чем ее особенность?

BlockCollection — это класс параллельной коллекции в C#, который обеспечивает безопасность потоков. Это означает, что несколько потоков могут одновременно добавлять и удалять объекты из BlockingCollection.

BlockingCollection<T> часто используется для реализации производитель-потребитель, где один поток добавляет элементы в коллекцию, а другой поток извлекает их для обработки.

1. **Ограничивающий:** Можно установить максимальное количество объектов, которые мы можем хранить в коллекции. Когда поток производителя достигает максимума, он блокируется для добавления новых объектов. На заблокированной стадии поток производителя переходит в спящий режим. Он разблокируется, как только потребительский поток удалит объекты из коллекции.
2. **Блокировка:** Когда BlockingCollection пуст, потребительский поток блокируется до тех пор, пока поток производителя не добавит новые объекты в коллекции.

В конце концов, поток-производитель вызовет метод CompleteAdding() класса BlockingCollection. Метод CompleteAdding() задает свойству IsCompleted значение true. Поток потребителя внутренне отслеживает свойство IsCompleted на предмет наличия элементов для использования в коллекции.

Основные особенности BlockingCollection<T>:

**Потокобезопасность**: Обеспечивает потокобезопасное добавление и извлечение элементов из коллекции.

**Блокирование при пустой/переполненной коллекции:** Методы Add() и Take() блокируют вызывающий поток, если коллекция пуста или переполнена соответственно. Это полезно для организации обмена данными между потоками, когда один из них ждет доступа к данным.

**Поддержка отмены операций:** Может работать с CancellationToken, что позволяет отменять ожидание при необходимости.

18. Как используя *async* и *await* организовать асинхронное выполенение метода?

**Асинхронность** позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы и при этом более экономно использовать потоки. Асинхронные методы выполняются в отдельных потоках. Однако при выполнении продолжительной операции поток асинхронного метода возвратится в пул потоков и будет использоваться для других задач. А когда продолжительная операция завершит свое выполнение, для асинхронного метода опять выделяется поток из пула потоков, и асинхронный метод продолжает свою работу.

Ключевыми для работы с асинхронными вызовами в C# являются два оператора: **async** и **await**, цель которых - упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода.

**Асинхронный метод** обладает следующими признаками:

* В заголовке метода используется модификатор **async**
* Метод содержит одно или несколько выражений **await**
* В качестве возвращаемого типа используется один из следующих:
  + Void, Task, Task<T>, ValueTask<T>

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами **out**, **ref** и **in**.

Также стоит отметить, что слово **async**, которое указывается в определении метода, НЕ делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений **await**.

